Automating Parallel and Distributed Quantative Stochastic Simulation

Victor Yau

September 17, 1996

Abstract

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Basic Problems in Quantitative Stochastic Steady State Simulation

Over the last decade stochastic simulation of telecommunication networks has become one of the most commonly used tools for their performance modelling and evaluation. This is the result of significant achievements of electronic and computer engineering, that have led to broad proliferation of powerful and cheap computers, and significant achievements in software technology, that resulted in very simple and efficient interactive human-computer interfaces. Today, it is natural for telecommunication engineers to study processes occurring in data communication networks by watching their animated, dynamic graphical representations on screens of monitors, using data generated by computers during simulation runs. There is easy access to various user-friendly simulation packages in which traditional discrete-event simulation modelling is supported by various concepts of artificial intelligence; see for example [DIRE90]. Some of these packages totally release users from burdens of programming, allowing them to construct simulation models from typical components appearing on screen in their iconic representations.

The pace of software inventions being introduced into ordinary practise has left behind developments in the area of simulation output data analysis in such an environment. Any simulation in which simulated events are functions of (pseudo)random numbers is simply a statistical experiment, and, as in any statistical experiment, simulation output data (observations collected during simulation runs) have to be properly, statistically analysed before any constructive conclusions about the investigated systems or networks are made. As stated by J. Kleijnen, a world authority on stochastic simulation, ”... computer runs yield a mass of data but this mass may turn into a mess ...” if the random nature of output data is ignored” and then ”... instead of an expensive simulation model, a toss of the coin had better be used...”, [KLEI79]. It is generally accepted that various software ”attractions”, such as graphical data representation and animation of simulated processes on computer monitors, are attractive and useful methods for validating simulation models and increasing the user-friendliness of simulation packages, but they don't substitute the need for a sound statistical analysis of collected data, including an assessment of statistical errors of the final estimates. There is no reason for avoiding this standard engineering practise during simulation studies. Unfortunately, the lack of a general method for analysing simulation output data and the complexity of the associated theoretical problems have led to an alarming situation in the field of performance evaluation of computer and electrical engineering systems, including telecommunication networks, in which the credibility of many published results of simulation studies can be questioned. As B.Gaither, the Editor-in-Chief of the ACM Performance Evaluation Review, has recently stated in his editorial: ¡there is not¿ ”... any other field of engineering or science where similar liberties are taken with empirical data ...” [GAIT90].

This is the situation that prompted us to investigate the possibility of a method for fully automating the control of stochastic steady state simulation experiments.

1. At what stage during the simulation should data collection begin ?

2. When should the simulation be stopped ? In other words, what duration of simulated time needs should the system of interest be simulated, before stopping the simulation ?

3. How can the system be simulated for the desired length of simulation time within a practical length of real time ?

These are the questions encountered during the programming and running of a simulation which should be answered by the method with a minimum of user involvement.

In seeking answers to questions of run length control, one must address the problems that arise because of the fact that observations collected from the simulated processes are usually correlated, therefore classical statistical techniques cannot be adapted for inferring the precision of estimates obtained during a simulation run. In addition, the simulated processes usually pass through an initial transient phase. Observations collected during the initial transient period neither belong to a stationary sequence nor do they characterise the steady-state behaviour of the simulated process.

In view of the goal of obtaining statistically reliable estimates from the simulation within a practical period of real time, and to direct our efforts, it is helpful to re-formulate the above questions as:

1. How can we determine when the simulated process has reached state ?

2. How many observations are needed to form an estimate(s) which has a required level of precision ?

3. Can the number of observations be reduced, for instance by using a more efficient estimator ? Alternatively, how can we reduce the time required to generate the required number of observations ?

see Fig. 1.1.

1.2 Structure of the Solution Explored in this Report

The work of this report was conducted in five stages:

Stage 1 Survey of basic problems and solutions of automatic sequential precision control of steady-state estimates.

Figure 1.1: Narrowing down the analytical problems of quantitative steady-state simulation to be considered in this report.

Stage 2 Comparative studies of the more promising methods for automatic precision control of estimates of steady-state mean values. The methods we evaluated are those that were selected during Stage 1 for their potential to reliably automate the data collection, statistical analysis, and precision and runlength control tasks during a simulation experiment.

Stage 3 Survey existing techniques for applying parallel and/or distributed processing in quantitative stochastic simulation. Draw insights of significance to Stage 4.

Stage 4 Development of SA-PTS, a novel approach to parallel simulation that employs a generalisation of the best method of sequential estimation suggested in Stage 2.

Stage 5 Software implementation and performance evaluation of the a parallel simulation package for the estimation of mean values in steady-state simulation experiments following the SA-PTS methodology of Stage 4.

An overview of the organisation of this report is shown in Fig. 1.2. The rationale and goals of each stage, and the chapters in which their results will be presented, will be overviewed in the remainder of this section.

Figure 1.2: The main stages of our research, and their order of presentation.

The survey of possible solutions to two major analytical problems of quantitative steady-state simulation, namely designing efficient and reliable rules for :

1. stopping steady-state simulations, when sufficient number of observations have been collected to form an estimate with the required level

of precision, and

2. starting steady-state analysis, when the initial transient period expires,

is contained in Chapter 2 of this report. We concentrated our attention on steady-state simulations aimed at estimating such cumulative measures of performance as mean values or probabilities of events; for example: the mean delay experienced by packets in a telecommunication network, or overflow probabilities of buffers in a network.

Comparative studies of different methods have not been satisfactory advanced yet. Relatively few of the methods had their accuracy analysed and none of those studies could be considered as an exhaustive one. In the final section of Chapter 2 we present the results of an empirical analysis of a few methods of precision control, selected by us as potential candidates for implementation. These are results of our exhaustive studies, reporting the results obtained by executing 2000 statistically independent replications of each experiment, at each working point. The main performance measures considered were :

- Coverage, which is a measure of the accuracy of the method,
- Run length, defined as the total number of observations that were generated, in order to produce the estimate. The run length indicates the computational costs of obtaining the estimate.
- Mean final relative precision of the estimate. The precision of the estimate at stopping point indicates the level of granularity offered by the method.

The methods evaluated are known as *sequential precision control methods* since they check the precision of estimates during consecutive points in a simulation run, and command the simulation to continue until the required level of precision is obtained.

At this juncture, let use note that another mode of conducting quantative stochastic simulation experiments, know as *fixed sample sized methods*, is also commonly used. An experiment conducted according to a fixed sample sized procedure is stopped unconditionally when a fixed number of observations have been collected. Fixed sample sized procedures are simplier to implement, and are useful if the available real/computing time is limited. Most methods considered herein can also be adapted for infering the precision of point estimates obtained by fixed sample sized experiments. Similarly the parallel simulation methodology proposed by us for speeding up the simulation process can be adapted to reduce the amount of real time required to perform a fixed sample size simulation experiment. In contrast to a simulation conducted under the sequential precision control methods, a simulation conducted according to a fixed sample sized method allows us to only estimate the accuracy of the point estimate, but not to produce a point estimate with a desired level of accuracy. Thus we will focus on sequential precision control methods in this report.

Sequential precision control is the specially important in steady-state simulations, conducted for studying systems' behaviour after a long period of time [PAWL90]. Unfortunately, estimating parameters of interest to a required level of precision is often computationally intensive and can require very long runs in order to obtain reliable final results, even in the case of moderately complex systems. Excessive run-times hinder the development and validation of simulation models, and may even totally inhibit some performance evaluation studies. The problem of long run-time is often multiplied because simulation is a step in the research loop, and often several 'pilot runs' are required before a production run. In addition, once results are at hand, the knowledge gained may raise new questions and hence the need for new runs with the simulator for a new region of the design space, or even the focus of the study is changed, requiring major changes and re-running.

Fortunately, modern multiprocessor and distributed computer systems offer high potential processing power that can be employed in parallel simulation, by applying *true parallel processing*[1] for speeding up simulation runs. The problem is that the research in parallel processing, having successfully solved many related problems, has not led yet to an effective tool for automated parallel quantitative stochastic simulation.

[1]As opposed to concurrent processing in which multiple tasks may be executed by one processor in an interleaved manner

Given a multiprocessor, or a network of workstations, a niave approach to using their joint capacity concurrently, is to run several simulations in parallel, one per machine. For example, if we wish to simulate ten CA-STAR networks, each differing in network size (number of stations), the simplest way to speedup the simulation process is to run one simulation per workstation. Nevertheless, there are reasons why this approach would generally be unattractive. One is that the execution time for each simulation typically varies, because the number of observations required to form an estimate of fixed precision is usually different for each simulated network. For this reason, speedup is typically a small fraction of the number of workstations used, since a subset of simulations will take the longest time (the bottleneck runs), whilst workstations running the shorter simulations will be idle once finished. This problem is very evident from the runlengths needed for estimating the mean waiting time in an M/M/1 queuing systems. It will be shown in Chapter 3 that a system with a normalised input load of 0.95 takes over one hundred times longer to execute, than the system with an input load of 0.10. The processing power utilization—if we were to run each of these ten cases on a separate workstation— would be less than 20%. Obviously a related drawback of this approach is that the degree of parallelism is limited by a number of simulations we wish to run. In the above example, we would be able to use at most ten workstations, even if twenty were available, thus further lowering the utilization of available processing power.

Another problem with running different simulations simultaneously for speeding up runtime, is that often we need to know the results from one simulation run before we could decide which region of the design space to simulate next. In this case, only one machine can be productively employed at a time.

Finally, the probability that a simulation run is affected by machine failure is proportional to its runtime, and running several different simulations, one per machine, would not speedup the execution time of each individual simulation. Hence the most time critical runs (longest runs) face the highest risk of failure.

An alternative approach to alleviating the problem of the long run-time needed to get reliable results, is to use a variance reduction techniques (VRT) if one can be applied. Still, many VRTs require special *a priori* knowledge of the system being simulated, and the practitioner has the burden of encoding that knowledge into a simulation in a way which yields speedup. However, the main drawback with VRTs as the main simulation speedup tool is that they are not universally applicable, and the extent to which they could reduce the required simulation execution time is strictly limited and depends on the system being simulated.

In view of these limitations, there has been growing interest in parallel or distributed stochastic simulation, where multiprocessors of a network of workstations are used in parallel to speedup the execution of one simulation task. Two different scenarios of parallel processing are possible, based either on:

- *single replication in parallel* (SRIP), where many processors and/or computers cooperate in running a single replication of a simulated system, or on
- *multiple replications in parallel* (MRIP), where processors are engaged into running their own, independent replications of the simulated system, but cooperating with central analysers of the simulation output data.

Research in parallel simulation has been almost entirely focused on SRIP, as an interesting problem of parallel processing; see such surveys as [MISR86], [KAUD87], [FUJI89], [WAGN89]; while MRIP has been considered only in a few publications despite that this is an attractive approach from the point of view of statistical methodology; see [HEID86], [HEID88], [GLYN91], [REGO91], [SUND91] and [REGO92]. It is obvious that the effectiveness of SRIP depends on the level of inherent parallelism existing in the system simulated. If this level is high, then SRIP can significantly speed-up execution of the simulation; of course if the problems of synchronisation, deadlocks and causality errors are properly solved. But, SRIP experiences a new problem: statistical analysis of simulation output data may require the data to be ordered according to one reference time (of the simulated model); this can be costly to achieve when data are collected from subprocesses executed asychronously in parallel. This problem does not exist in MRIP, since executions of single replications are not parallelized, thus data collected within each replication are always properly ordered in time. Additionally, considering the statistical properties of estimates from SRIP and MRIP, it is possible to show that the latter scenario is statistically more efficient than the former, in the sense of the mean squared error of final estimates, if the problem of the initialization bias is effectively solved [HEID86]. By this reason our studies have been restricted to stochastic simulations following MRIP scenario.

The two basic approaches of parallel stochastic simulation: SRIP and MRIP, as well as their possible programming and runtime environments are discussed in Chapter 4. This includes the method of Spectral Analysis in Parallel Time Streams (SA-PTS), that, based on our experience with sequential precision control of simulations executed on a uni-processor, has being proposed by us for simulations executed according to the MRIP approach.

Our main motivation for developing a user-friendly package for parallel steady-state stochastic simulation was to overcome problems caused by long simulation times experienced in our on-going research in performance evaluation of high-speed and integrated-services communication networks, and WDM local and metropolitan area networks. Thus the need to maintain basic statistical rigours of proper analysis of simulation output data experienced in our on-going research in performance evaluation of telecommunication networks, combined with the drawback of long simulation runtimes led us to the development of a parallel simulation package designed for solving both problems. It is intended for use on multiprocessor systems and/or heterogeneous computer networks, involving an arbitrary numbers of processors.

The main design goals and the user programming and runtime interface of such a package are addressed in Chapter 5. It was assumed that the package should accept ordinary (non-parallel) simulation programs, and all further stages of stochastic simulation should be transparent for users. Such a package should automatically transform sequential simulators into ones suitable for parallel execution. At runtime, the parallel simulation would exist as a set of cooperating parallel processes, possibly executing on several machines inter-connected by a local area network. The control of the precision of estimates, and stopping of all parallel replications and global precision control processes when the required precision of all steady-state estimates have been achieved should be performed transparently. These ideas, together with mechanisms for efficient distributed process creation, inter-process communication, process management, and distributed termination were materialised in AKAROA (a simulation p**a**c**k**age for **a**utomatic generation and cont**r**ol of processes for parallel st**o**chastic simul**a**tion)[2].

AKAROA consists of two main modules: Parallel Simulation Manager, responsible for the automatic creation of parallel simulation and global precision control processes, process management, and interprocess communication; and Control, responsible for controlling simulation run-time and analysis of output data collected during MRIP-type steady-state simulation. The package, as described in Chapter 5, is also equipped with Build, a module which can be used for rapid construction of typical simulation models. Thus, AKAROA consists of three basic modules, which create one object-oriented package for MRIP-type stochastic simulation, written in C++. Users have access to services offered by AKAROA through a simple programming interface, see Fig. 1.3. At run-time the Parallel Simulation Manager of AKAROA cooperates with the user runtime interface process of AKAROA to present the multiprocessor and/or network of workstations as one (virtual) uni-processor to the user. The architecture, implementation, and the results of initial performance studies of AKAROA are reported in Chapter 6.

Chapter 7 summaries the main results of this report, and gives our insights.

In addition to this manuscript, details of our research and examples of some of its applications can also be found in the following reports.

[PAWL94] Pawlikowski K., Yau V., and McNickle D., "Distributed stochastic discrete-event simulation in parallel time streams", Proc. 1994 Winter Simulation Conference, IEEE Press, pp.723-730

[YAU93] Yau V. and Pawlikowski K., "AKAROA: a package for automating generation and process control of parallel stochastic simulation", in proc. of the Sixteenth Australian Computer Science Conference, ed.G.Gopal et al., Australian Computer Science Communications, 1993, pp71-83.

[YAU92] Yau V. and Pawlikowski K., "On automatic partitioning, runtime control and output analysis methodology for massively parallel simulations", Proc. European Simulation Symposium ESS 92, pp.135-139, Dresden, Germany, Nov 1992.

[2] Also a picturesque spot on Banks Peninsula in the South Island of New Zealand

Figure 1.3: The Block Structure of AKAROA

[PAWL92] Pawlikowski K., Yau V., An empirical comparison of sequential estimators for output data analysis in steady state simulation of high speed data networks", in proc. Operations Research Society Conference'92, pp175-177, Christchurch, New Zealand, 1992.

[YAU92] Yau V. and Pawlikowski K., "A class of protocols for heavy loaded multiple-channel local area networks", in proceedings IEEE/ACM International Conference on Communications ICC'92, pp 742-748, July, 1992, in Chicago, U.S.A.

[YAU92] Yau V and Pawlikowski K., "Improved Nested-Threshold-Cell-Discard buffer management mechanisms", in proc. IEEE Int. Conf. on Computers, Communications, and Automation, TENCON'92, (Melbourne, Australia, Nov1992), IEEE Comm.Press, 1992, pp 820-824.

[YAU92] Yau V. and Pawlikowski K., "ATM Overload Control: Nested Threshold Cell Discarding with Suspended Execution", in proc. of the Australian Broadband Switching and Services Symposium, ABSSS'92, pp 689-706, Melbourne, 1992.

[PAWL91] Pawlikowski K., Yau V., "Independent replications versus spectral analysis in steady-state simulation of high speed data networks", in proc. ATRS'91, Nov. 1991, Wollongong, Australia, pp 182-190.

[YAU96] Yau V., and Pawlikowski K., "A Conflict-free Traffic Assignment Algorithm using Forward Planning," IEEE INFOCOM'96, vol.3.

Pawlikowski K., McNickle D., and Yau V., "Object-oriented model construction, and automated distributed simulator generation and output analysis ", Final R&D report, Australia Overseas Telecom Corporation (AOTC Telstra), Con. 7314, Also, technical report no. COSC 02/93, Dept. of Computer Science, University of Canterbury.

Pawlikowski K., and Yau V., "Methodology for stochastic simulation for performance evaluation of data communication networks", Final Report for Telecom Corporation of New Zealand, Wellington, New Zealand. Also Technical report no. COSC 03/93, Dept. of Computer Science, University of Canterbury.

Yau V. "Polarised Independent Replications: An alternative approach to estimating steady state parameters by simulation in the presence of an initialisation bias", in preparation.

Chapter 2

Steady-state Simulation Experiments: Basic Problems and Solutions

Obtaining statistically valid results by simulation is difficult due to the fact that observations collected during simulations are typically correlated, and that the simulated process initially moves along a non-stationary trajectory.

Consider the sequence of observations $x_1, x_2, ..., x_n$ collected during a simulation run. The observations can be used to estimate sample mean μ_x by calculating the arithmetic average of the sample:

$$\overline{X}(n) = \sum_{i=1}^{n} \frac{x_i}{n} \tag{2.1}$$

But, let us note that this estimate is a function of the sequence of random observations x_1, x_2, ..., x_n, and, as such, it assumes different, random values in different simulation experiments. Following standard statistical approach, the accuracy of any such estimate can be assessed by considering the probability

$$P(\overline{X}(n) - \Delta_x \leq \mu_x \leq \overline{X}(n) + \Delta_x) = 1 - \alpha, \tag{2.2}$$

where

$$\Delta_x = t_{df,1-\alpha/2}\hat{\sigma}[\overline{X}(n)], \tag{2.3}$$

$t_{df,1-\alpha/2}$ is the $(1-\alpha/2)$ quantile of the Student t-distribution with df=n-1 degrees of freedom, and $\hat{\sigma}[\overline{X}(n)]$ is the estimator of the variance of $\overline{X}(n)$. Thus, the accuracy of $\overline{X}(n)$ is determined here by its *precision* Δ_x, or the half-width of the confidence interval , at an assumed confidence level (1-α), $0 < \alpha < 1$. Let us note that Eqn 2.2 also says that with probability α the interval $(\overline{X}(n) - \Delta_x, \overline{X}(n) + \Delta_x)$ does not contain μ_x (!), or that repeating the simulation a number of times the (unknown) value μ_x would be outside the interval in 100α% of cases.

It is well known that if the observations x_1, x_2, ..., x_n can be regarded as realisations of *independent and normally distributed* random variables X_1, X_2, ..., X_n, then the unbiased estimator of the variance of $\overline{X}(n)$ is

$$\hat{\sigma}^2[\overline{X}(n)] = \frac{1}{n(n-1)}\sum_{i=1}^{n}(x_i - \overline{X}(n))^2 \tag{2.4}$$

and $t_{df,1-\alpha/2}$ is the (1-α/2)quantile in the Student t-distribution with df=n-1 degrees of freedom. The formula (Eqn. 2.4) can be also applied when the observations x_1, x_2, ..., x_n are not drawn from a normal distribution, if they represent *independent and identically distributed* (i.i.d.) random variables X_1, X_2, ..., X_n. In such a case, by the virtue of the central limit theorem, an acceptable approximation could be obtained when the number of collected observations was sufficiently large (n >100).

Unfortunately, observations collected during practical simulations are not statistically independent, since they appear to be strongly autocorrelated. The general formula for the variance of the mean $\overline{X}(n)$ of observations x_1, x_2, ..., x_n collected from a *covariance stationary* process is

$$\hat{\sigma}^2[\overline{X}(n)] = \left[R(0) + 2\sum_{k=1}^{n-1}(1 - \frac{k}{n})R(k)\right]/n \tag{2.5}$$

where

$$R(k) = E[(X_i - \mu_x)(X_{i-k} - \mu_x)], \quad 0 \le k \le n-1 \tag{2.6}$$

is the autocovariance of order k (the lag k component of the autocovariance function of the process), but its direct application in sequential simulation would still lead to significant inaccuracies, caused by the initial non-stationarity of the process from which the observations were collected, and difficulties with estimating higher order autocovariances. Neglecting the existing statistical autocorrelations would be equivalent to ignoring all the terms except R(0) in Eqn. 2.5. It could lead to significant errors of estimation. For example, in an M/M/1/∞ queuing system with 90% utilization, the variance of the mean queue length calculated according to Eqn. 2.5 is 362.6 times greater than that from Eqn. 2.4 [BLOM67]. Estimating $\sigma^2[\overline{X}(n)]$ without regard for the autocorrelation among the observations would lead to either an excessively pesimistic confidence interval for μ_x, in the case of negatively correlated observations, or to an excessively optimistic confidence interval for μ_x in the case of positively correlated observations. A positive correlation between observations is typical in simple queuing systems without feedback connections, and it is usually stronger for a higher system utilization. The estimation of the variance of the sample mean in autocorrelated processes is a complex statistical problem, and therefore it is also a major problem in assessing precision of the mean value during stochastic simulation. Possible solutions to the problem of autocorrelated observations, and their suitability for implementations in the Control module of AKAROA which is responsible for automatically stopping a parallel simulation when the precision of the final results reached a proper level, are discussed in section 2.1.

An additional problem, specific for steady-state simulations, is caused by the fact that simulated stochastic processes initially move along their non-stationary trajectories, thus changing their statistical properties with time. Observations collected during the initial transient period[1] neither belong to a stationary sequence nor characterise steady-state behaviour of the simulated process. Neglecting the existence of initial transient periods can lead to significant bias in steady-state estimates of analysed performance measures. The solution to this problem, a technique for automatic detection of the end

[1] In simulation jargon, this period of time is also called a "warm-up" period or the "transient phase"

of initial transient period, has been implemented in AKAROA, and will be presented in section 2.2.

2.1 Precision control of steady-state mean values

Recently it has become generally accepted that the only proper approach to controlling precision of the final estimates during simulation studies is a sequential one, based on assessing the precision of estimates at consecutive checkpoints, as long as the precision of analysed performance measures is not satisfactory. The most useful criterion for stopping steady-state simulations is based on comparing the current value of *the relative precision* of the estimate with its maximum acceptable value. The relative precision of an estimate is defined as the relative half-width of its confidence interval (at a given confidence level), i.e. as the ratio

$$\epsilon = \frac{\Delta_x}{\overline{X}(n)} \tag{2.7}$$

$0 < \epsilon < 1$; where Δ_x, the current value of the half-width of the confidence interval, is calculated from Eqn. 2.4. The simulation is stopped at the first checkpoint at which $\epsilon \leq \epsilon_{max}$, $(0 < \epsilon_{max} < 1)$, where ϵ_{max} is the assumed, maximum acceptable value of precision of the results.

A number of different methods has been proposed to analyse the precision of mean value estimates, or equivalently, to estimate the variance of estimators of mean values, in steady-state simulation. They can be grouped into the following classes:

- methods of independent replications,
- methods of polarized independent replications,
- methods of batch means,
- methods of spaced batch means,

- methods of overlapping batch means,
- methods based on spectral analysis,
- methods of regenerative cycles,
- methods of uncorrelated sampling,
- methods based on standardised time series, and
- methods based on autoregressive representation.

These methods differ from each other because they apply different approximations and data transformations for determining the confidence interval of the mean. Hence the quality of the final point and interval estimators produced may vary depending on the choice of method. The quality and robustness of each method can be assessed by its *coverage*, the empirical frequency with which the final confidence intervals produced by a method $(\overline{X}(n) - \Delta_x, \overline{X}(n) + \Delta_x)$ contain the true parameter μ_x, at a given confidence level $(1 - \alpha)$ $0 < \alpha < 1$. Of course, coverage analysis can be applied only to systems with theoretically well known behaviour, since the value of μ_x has to be known. It requires a statistically significant number of repeated simulation experiments (usually 200 or more replications are performed) to determine the fraction of experiments which produced a final confidence interval that covers the true mean value of the estimated parameter. A given method is considered as producing valid $100(1 - \alpha)\%$ confidence intervals (for, say, the mean delay) if the upper bound of the confidence intervals for the coverage is at least $(1 - \alpha)$. Otherwise, the method should be regarded as inaccurate.

Comparative studies of different methods have not been satisfactory advanced yet. Only relatively few of them had their coverage analysed and none of those studies could be considered as an exhaustive one[2]. In the next

[2]There are claims that there is no theoretical basis for extrapolating results found for simple, analytically tractable systems to systems of more complex structures, which are the subjects of real simulation studies; see [FOX78]. On the other hand, there is also no other practical way of assessing quality of the methods, and if a method behaves well in the case of a (very) dynamically active stochastic system, such as M/M/1/ queue, one should expect that it would behave at least as well in the case of stochastic systems occurring in the real world, that have typically weaker dynamics.

chapter we present the results of coverage analysis of several methods of precision control, selected by us as potential candidates for implementation in the Control module of AKAROA. These are results of our exhaustive studies, reporting the results obtained by executing 2000 statistically independent replications of each experiment, at each working point.

The quality of the methods for studying confidence intervals of mean values can be also assessed theoretically. For example, some theoretical results on the coverage error and its main sources were presented in [GLYN82], [KANG85] and [SCRU80]. Asymptotic properties of different variance estimators $\hat{\sigma}[\overline{X}(n)]$ and the limit values (as $n \to \infty$) of their bias

$$Bias\hat{\sigma}[\overline{X}(n)] = E\hat{\sigma}[\overline{X}(n)] - \sigma[\overline{X}(n)] \qquad (2.8)$$

and the variance $\mathrm{Var}[\hat{\sigma}^2[\overline{X}(n)]]$, where $\hat{\sigma}^2[\overline{X}(n)]$ is the estimator of $\sigma^2[\overline{X}(n)]$, were studied in [GOLD85] and [GOLD86]. Alternatively, Schmeiser [SCME82] proposed studying the asymptotic properties of the expected values and variances of the half-width of confidence intervals Δ_x generated by a method; see also [GLYN85], and [GOLD84]. Following such criteria one could say that the method using the variance estimator with the smallest bias and smallest variance, or using the estimator of the width of confidence interval having the smallest expected value and smallest variance, is (asymptotically) superior to others. Unfortunately these criteria are not universal, since a small bias can be accompanied by large variance, or vice versa. In terms of confidence intervals it can mean wide and very variable confidence intervals giving good coverage, or, conversely, stable and narrow confidence intervals giving poor coverage.

A few other measures of the effectiveness of the methods were proposed in [SCRI81]. Below we survey the above-listed methods from the point of view of their applicability in the control module of a parallel simulation package that would automatically start and stop simulation processes when the precision of the final estimates reaches the required level, to select a method that will be implemented in a simulation package to be used by users with very little knowledge of the estimation theory and statistics, assuming also that very little is known about the dynamics of the simulated processes.

2.1.1 Independent Replications

The methods of Independent Replications (IR) overcome the analytical problem created by the autocorrelated nature of the original output data in a conceptually simple way: the simulation is repeated a number of times, each time using different, independent sequence of random numbers. Observations collected during a replication are used to obtain only one, secondary data point: the average value over observations collected during this replication. The set of averages are used in further statistical analysis as evidently independent and identically distributed output data.

However, the average of the observations in each replication may be a strongly biased estimator of the steady state mean, because observations collected during the initial transient period of each replication may not be representative of observations from the process in its steady state. In the context of IR, we can identify three approaches for dealing with the problem of initial transient bias. The first approach is to estimate the length of the initial transient period, and discard data collected during the transient period. The argument in support of this approach is that it should reduce the bias of the estimator, and hence improve its quality in terms of its coverage. The second approach is to ignore the presence of an initial transient period, and assume that all data were collected from the process in its steady state. The argument in support of this approach is that, by not discarding any observations, more observations are available for producing the estimate, so the variance of the estimator, and hence its relative precision should be improved. The third approach, developed by us, is to directly consider the bias in the estimator, and also refrain from discarding any observations. According to this approach, either the interval estimate for the mean is adjusted to accounted for the estimated bias, or the length of each replication is increased until it has being determined that the biased caused by observations collected during the transient period is small, relative to the variance of the esitmator [YAU96].

In practically all reported cases when sequential versions of IR were applied in conjunction with initial data deletion (the first approach to dealing with the initial transient), the length of replications and the number of discarded initial observations had to be predicted in advance and kept constant, while the number of replications was adjusted dynamically.

Having generated k_r replications, with m observations collected during each of them, we have k_r sequences of data, (x_{11}, x_{12}, ..., x_{1m}), (x_{21}, x_{22}, ..., x_{2m}), ..., (x_{k_r1}, x_{k_r2}, ..., x_{k_rm}). Now n_0 initial observations are removed from each replication. The replication means are then calculated as :

$$\overline{Y}_i = \overline{X}_i(m - n_{0i}) = \frac{1}{m - n_{0i}} \sum_{j=n_{0i}+1}^{m} x_{ij} \quad i = 1, 2, ..., k_r \tag{2.9}$$

The secondary data, Y_1, Y_2, ... , Y_{k_r}, can be considered as realisations of i.i.d. random variables that can be used to get the point and interval estimator of μ_x by substituting n by k_r, x_i by $\overline{Y}$, and $\overline{X}_n$ by

$$\overline{\overline{X}}_{IR} = \frac{1}{k_r} \sum_{i=1}^{k_r} \overline{Y}_i \tag{2.10}$$

in Eqns. 2.1 to 2.4. Authors of such sequential simulations tried to find the best trade-off between the number of replications and their length, for achieving good quality of the final estimators. For example, at least 100 final observations in each replication (having discarded initial observations) were suggested to secure normality of the replication means [FISH78, p.122]. Moreover, as shown in [LAW77] and [KELT84], it is better to keep replications longer than to make more replications, since it would usually improve the final coverage too.

In a fully sequential version of IR using discarding, both the length of initial transient period and the length of each replication should be determined dynamically during simulation runs. The first step in this direction has been recently proposed in [PAWL91], with different numbers of initial observations discarded during different replications (following a sequential, independent estimation of the length of initial transient length) but with a fixed total replication length in steady state, i.e. having deleted initial observations. Full automation of this method would additionally require making the length of each replication related to the dynamic characteristics of the simulated process. It could be done by making the length of each replication related to the length of its initial transient period. But, such an enhancement of IR creates a statistical problem. Namely, the variance of the global mean in such scenario would be a weighted sum of replication means calculated over different sample sizes. It requires special precautions to be taken for avoiding worsening the quality of such global estimators. This effect has not been studied yet, neither theoretically nor empirically.

As mentioned, the second application of IR is to ignore the possibility that the initial observations of each replication may have being generated whilst the process was in a non-stationary state. The point and interval estimator under this approach are the same as that for IR with discarding described above, except that n_{0i} (the number of initial observations discarded per replication) would be zero.

Since our initial results of the coverage analysis of IR with discarding indicated good quality of this method, [PAWL91], [PAWL92], we have included this technique in our comparative studies, the results of which are reported in Chapter 3. Unfortunately, IR does not seem to be a good candidate for sequential simulation under time constraints, for example when executed on single sequential computer, since it may require prohibitively long time of execution caused by discarding initial observations (collected during the initial transient periods) at the beginning of each replication. It can lead to a poor utilization of the total simulation time if the lengths of initial transient periods are significant.

2.1.2 Polarized Independent Replications

Instead of alleviating the problem of initialzation bias by deleting initial data (first approach), or assuming that the initialzation bias is insignificant (second approach), it may be of interest to directly consider the bias in the estimator, and account for it in the interval estimate for the mean, or to incrementally increase the length of each replication until it has being determined that the biased caused by observations collected during the transient period is a statistically insignificant component of the mean squared error (MSE) of the estimator [YAU96].

The method following this approach is called Polarised Independent Replications. Instead of launching all replications from the same initial state, the method of Polarised Independent Replications launches half of the replications initialised to the empty-and-idle (-ve) state, and half of them from the full-and-busy (+ve) state. At various checkpoints, the average of the observations from all replications are collected to form a point estimate of the mean of interest (e.g. mean throughput). Polarised Independent Replications depends on the assumption that estimates obtained from replications launched from the +ve state have a positive or zero bias, and estimates obtained from replications launched from the -ve state have a negative or zero bias. Then a point estimate of the maximum value of the bias of the estimator of the mean can be obtained from the difference between the average of results of replications launched from the +ve state, and the average of results of replications launched from the -ve state. If the estimate has reached the required level of precision, and if the bias of this estimate is statistically insignificant compared to the standard deviation of the estimator (standard error), then the simulation could be stopped.

The quality of estimates produced using Polarised Independent Replications has yet to be empirically studied. However Polarised Independent Replications does not seem to be a good candidate for parallel execution on an multiprocessor, or on a network of machines, as will be shown in Chapter 3.

2.1.3 Methods of Batch Means

The methods of batch means (BM) that have been proposed can be categorised into three classes:

- non-overlapping batch means (NOBM),
- overlapping batch means (OBM), and
- spaced batch means (SBM).

All of them require that sequences of analysed data are stationary, thus initial observations, collected during initial transient periods, should be discarded following, for example, the method discussed in the next section. Then, "steady-state" observations are collected during a single (long) simulation run, and for weakening correlations existing between consecutive data, the recorded sequence of n original observations $x_1, x_2, ..., x_n$ is divided into nonoverlapping batches $(x_{11}, x_{12}, ..., x_{1n})$, $(x_{21}, x_{22}, ..., x_{2n})$, ... of size m, sufficiently large for making mean values over these batches (almost) independent.

Thus, mean values over consecutive non-overlapping batches of observations are used as (secondary) output data in the statistical analysis of simulation results. This approach is based on the assumption that observations more separate in time are less correlated; see [BRIL73] for a formal justification. By the central limit theorem [POLY20], the batch means should also be approximately normally distributed.

Selecting a batch size m that ensures uncorrelated batch means appears to be the major problem. A natural solution is to estimate the correlation between batch means starting from an initial batch size m_1, and, if the correlation cannot be ignored, increase the batch size and repeat the test. Thus, at this stage the method in its sequential version requires two procedures: the first responsible for sequentially testing for an acceptable batch size, and the second responsible for sequentially testing the accuracy of estimators. The sequence of batch means can be regarded as non-autocorrelated when the correlation coefficients of all lags assume small magnitudes; say, if they are less than 0.05. One can also determine the threshold for neglecting the autocorrelations in a statistical way, by testing their values at an assumed level of significance; see [ADAM83] and [WELC83, p.306].

One of the problems associated with estimating the correlation coefficients is that estimates of correlation coefficients of higher lags are less reliable since they can be calculated from fewer data points within the batch. Usually it is suggested to consider lags not greater than 25% of the sample size ([BOXJ70, p.33]) or even not greater than 8 -10% (c.f., [GEIS64]). Law and Carson [LAWC79] have proposed a procedure for selecting the batch size for processes with autocovariances monotonically decreasing with the value of the lag; see also [LAWK82]. In such a case only the lag 1 autocorrelation has to be taken into account. In the same class of processes one may also test batch means against autocorrelation using von Neumann's statistic, [VONN41]. Such an approach was applied in [FISH78a] to processes with positive autocorrelation which decreases monotonically with m. Its sequential implementation, together with the control variates variance reduction technique, was proposed in [ANON86]. The observations were batched not by count but by time, i.e., over equal time intervals, whose length was specially selected, giving an uncorrelated sequence of time means over the intervals. Generally, procedures proposed for selecting a proper batch size can employ various statistical techniques and various criteria, and the final size of batches is random; different batch sizes will be normally found even in different replications of the same process.

Some studies of batch means techniques reported poor coverage of batch means interval estimators when they are applied in simulation studies of heavy loaded systems. It is probably caused by the fact that sometimes too small batch sizes are accepted as sufficient for getting uncorrelated batch means. For example, the procedures proposed by Fishman in [FISH78a] and [FISH78, p.240] can select batches of as few as 8 observations. Law [LAW83] refers to simulation studies of M/M/1 queues in which the method of batch means with the procedure proposed in [LAWC79] was used. Using k_b=10 batches of size m=32, for system utilisation ρ=0.9, and 500 repeated simulation experiments, the achieved coverage of the nominal 90% confidence intervals was only 63%. For these reasons, Kleijnen et al. [KLEI82] suggested the use of a modified Fishman's procedure accepting batches at least 100 observations long, while Welch [WELC83, p.307] recommended constructing batches at least 5 times larger than the size m given by a test against autocorrelation, provided that at least 10 such batches can be recorded.

Schmeiser [SCME82] analysed theoretically the trade-off between the number of noncorrelated batches, the batch size and the coverage of confidence intervals. It was shown that usually the number of batches used in the analysis of confidence intervals should be not less than 10, and does not need be greater than 30, if the simulation run is long enough to secure an adequate degree of normality and independence of batch means. This means that having determined a batch size which gives negligibly correlated and approximately normal batch means (which can sometimes require even a few hundred batches to be tested), there is no need to use more than k_b=30 batches to obtain confidence intervals with a good coverage. Thus, confidence intervals should be more reliable if obtained from a small number of longer batches. It is obvious that such a transformation improves the normality and independence of batch means, and as such usually yields better coverage of the confidence intervals.

Non-Overlapping Batch Means

(NOBM) is the method of batch means that is most often used in practise (and usually called just "batch means"). As any method based on batch means it could be implemented relatively easily in a fully automatic simulation package that would totally free users from burden of statistical analysis of simulation output data, provided that the problem with initial transient period is successfully solved. This method has been already proposed in a few commercially distributed simulation packages, such as SIMSCRIPT II.5 and its specialised variations: Network II.5 and COMNET II.5, [MILS87].

If initial (transient) observations are discarded for satisfying the requirement of identical distributions of analysed data points, the NOBM interval estimator of the mean is obtained from Eqn. 2.4,by simply substituting consecutive observations by consecutive batch means, and using the Student t-distribution df=k_b -1 degrees of freedom ; where k_b is the number of batches.

The popularity of NOBM among practitioners continues regardless of reports of relatively poor coverage of this method, especially in heavy loaded systems; see e.g. [SCRI79] and [PAWL90a]. But, in this region only a little improvement can be achieved by using other methods.

NOBM has been selected as a candidate method for being the basis of a precision control method which could be implemented as part of the Control module of AKAROA, our package for automatic generation of parallel simulators and the control of steady-state simulation processes and output data analysis. The method for determining the batch size that gives uncorrelated batch means that we implemented (to conduct a full empirical comparison of selected methods), and the structure of the resulting batch means output analysis object will be presented in Chapter 3, and the C++ Class definition is contained in Appendix **??**. The results characterizing its performance, together with the performance of other candidate methods, are presented in Chapter 3.

Overlapping Batch Means

Overlapping Batch Means (OBM) has been proposed to overcome the problems with possible poor coverage of NOBM. The idea has been suggested by

Meketon and Schmeiser [MEKE84] for decreasing the variance of the variance estimator of mean and for increasing the number of its degrees of freedom. As mentioned, estimators with more degrees of freedom and smaller variance have shorter and more stable confidence intervals. Meketon and Schmeiser showed that such an aim can be achieved by overlapping batches of observations, having determined the batch size satisfying the requirements of NOBM. The point estimator of OBM assumes the same form as the NOBM one but its variance is calculated from a sequence of means of overlapped batches. The number of degrees of freedom depends on the degree of batch overlapping, and is greater from 1.5 to 1.33 times than the degrees of freedom in NOBM. The former characterises the maximum overlapping of batches (by m-1 observations), while the latter corresponds to batches overlapped by half. It is accompanied by variance reduction between 66% to 75%, respectively [MEKE84], [WELC87]. The amended rule for determining the number of degrees of freedom for OBM, when the number of batches is small, was suggested in [KANG90]. An important feature of OBM is that one can increase the number of batches within a given length of simulation run, without decreasing the size of batches. Asymptotic superiority of OBM over NOBM was shown theoretically; see [MEKE84], [DAME87], [KANG87], and [SCME90]. Its superiority in practical cases remains to be shown experimentally, yet. OBM seems to be an attractive candidate for an implementation as a part of AKAROA, despite being computationally more complex, although it remains O(n); see [MEKE84] and [SONG88]. It may require special data structures for speeding up overlappings, especially if different degrees of overlapping may be applied. Thus OBM has also being included in our comparative studies.

Spaced batch means

Spaced batch means (SBM) is yet another, recently proposed, method for estimating sample means based on the concept of (almost) uncorrelated batch means. Namely, for lower correlations between batch means it is proposed to separate batches by removing some observations between them [FOXG91]. In fact it is not a new idea, since the concept of spacers between observations was introduced already by [CONW63]. Spacers proposed in [FOXG91] are taken from original batches and they have size $\lfloor n^{\gamma} \rfloor$ where 0< γ <0.5, and n

is the number of observations collected. It is suggested to use the same point estimator for SBM as for NOBM, but for calculating the confidence interval one should use

$$\hat{\sigma}^2_{SBM}[\bar{\bar{Y}}] = \frac{m}{n^*(k_b - 1)} \sum_{j=1}^{k_b} (\overline{Y}_j - \bar{\bar{Y}})^2 \tag{2.11}$$

where for k_b original batches of observations of size m each, i.e. for $n{=}m_{k_b}$: $m^*{=}\ m$ - s, $n^* = k_b m^*$, and

$$\overline{Y}_j = \frac{1}{m-s} \sum_{k=1}^{m-s} x_{(j-1)m+k} \tag{2.12}$$

is the j-th spaced batch mean, while

$$\bar{\bar{Y}} = \frac{1}{k_b} \sum_{j=1}^{k_b} \overline{Y}_j \tag{2.13}$$

is the overall mean of spaced batches. The confidence interval of SBM estimator is approximated as

$$\overline{X}(n) \pm t_{k_b-1,1-\alpha/2} \hat{\sigma}^2_{SBM}[\bar{\bar{Y}}]. \tag{2.14}$$

Selected asymptotic properties of SBM are discussed in [FOXG91]. Let us note that the size of spacers could be also selected sequentially, applying the same tests as for determining the batch size in the NOBM. In the example considered in [SOLO83] the separating intervals of length s=25 were selected by applying the Spearman rank correlation test. No results on effectiveness of this method are available, but some regard that it wastes too many observations. Let us also note that we can be penalised for introducing spacers (to weaken inter-batch correlations) by increase of the variance of estimator. Thus, the usefulness of SBM for automating the statistical analysis of simulation output data has yet to be proved.

2.1.4 Spectral Analysis

Methods of estimation based on spectral analysis (SA) directly exploit the serial correlation between observations collected during one long simulation run. It is required that analysed observations represent a stationary and autocorrelated sequence $X_1, X_2, ..., X_n$, thus, as in previous methods of sequential estimation, we assume that initial observations, collected during the initial transient period have been discarded. The analysis of variance is shifted into the frequency domain by applying a Fourier transformation to the autocorrelation function $R(k)$, k= 0, 1, 2, ..., yielding the spectral density function

$$p_x(f) = R(0) + 2\sum_{j=1}^{\infty} R(j)cos(2\Pi fj), \tag{2.15}$$

for $-\infty \leq f \leq +\infty$. Because of the randomness of the collected observations, the spectral density function is a random function too. Comparing Eqn. 2.15 with Eqn. 2.5 one can see that for sufficient large n, the estimator of $\sigma^2[\overline{X}(n)]$ can be obtained from an estimator of p_x(f) at f=0, i.e.

$$\hat{\sigma}^2[\overline{X}(n)] \approx \frac{1}{n}\hat{p}_x(0) \tag{2.16}$$

Several techniques have been proposed for obtaining good estimators of the spectral density function p_x(f). Most of them follow classical techniques of spectral estimation, based on the concept of spectral windows (special weighting functions, introduced for lowering the final bias of the estimators), such as Tukey-Hanning or Parzen weighting functions; see [FISH78]. The usefulness of spectral windows in reducing the bias of the estimate $\hat{p}_x(0)$ has been questioned in [DUKE78], [WAHB80], [HEID81] and [HEID81a], although it has been shown that SA based on a modified Bartlett window functions is, in a statistical sense, asymptotically equivalent to OBM; see [DAME87] and [WELC87].

Another method based on spectral analysis for estimating $\sigma^2[\overline{X}(n)]$ was developed by Heidelberger and Welch ([HEID81], [HEID81a]). Their method, called here as SA/HW, estimates $p_x(0)$ from a regression fit to the logarithm of the average periodogram of the sequence of observations $x_1, x_2, ..., x_n$. The periodogram is a function of the discrete Fourier transforms $\{A_x(j)\}$ of the observations, i.e.

$$\Pi(\frac{j}{n}) = \frac{1}{n}\,|A_x(i)|^2 \tag{2.17}$$

and

$$A_x(i) = \sum_{s=1}^{n} x_s e^{-2\pi\iota(s-1)i/n} \tag{2.18}$$

where $\iota = \sqrt{-1}$. It can be shown that for 0< $j < n/2$

$$p_x(j/n) \approx= E[\Pi(j/n)] \tag{2.19}$$

To find an unbiased estimate of $p_x(0)$ the periodogram is transformed into a smoother function, namely into the logarithm of the averaged periodogram

$$L(f_j) = log\,\{[\Pi((2j-1)/n) + \Pi(2j/n)]/2\} \tag{2.20}$$

for f_j = (4j-1)/n. Next, this smoother function is approximated by a polynomial to get its value at zero.

We will refer to this method as SA/HW after its authors [HEID81]. An algorithmic description was provided in [PAWL90]. Its special feature is that the degrees of freedom of its t-statistic $\frac{\overline{X}(n)-\mu_x}{\hat{\sigma}[\overline{X}(n)]}$ do not depend on n, but on the way the collected observations are grouped when analysed. Despite the number of approximations involved, all experimental studies of SA/HW that we have recently performed within a wide spectrum of simulated systems show a good coverage of SA/HW estimators; c.f. [PAWL91], [YAU92], [PAWL92], [PAWL93a]. We are not aware of any theoretical studies on the quality of the SA/HW estimators yet.

A useful practical feature of SA/HW is that it can work with reduced data sets. Specifically, we can use a fixed number of (aggregated) output data points during the whole course of data analysis. Simply, if n observations are grouped into b batches of arbitrary size m , then for $n = bm$

$$\frac{p_x(0)}{n} = \frac{p_{\overline{X}(m)}(0)}{b} \tag{2.21}$$

where, for $-\infty \leq f \leq +\infty$,

$$p_{\overline{X}(m)}(f) = R(0,m) + 2\sum_{j=1}^{\infty} R(j,m)cos(2\Pi f j), \tag{2.22}$$

is the spectral density function of the autocorrelation function R(k,m) (k=0, 1, 2, ...) of the batch means over batches of size m. This insensitivity of SA/HW to batching the observations allows the batch size to be increased dynamically (starting from m=1), while keeping in memory only a limited number of the batch means. A special batching/rebatching procedure were proposed in [HEID81] and [HEID81a], see also [PAWL90]. It appears to be a very efficient way of limiting the required data memory space. This, and the encouraging results of our quality studies of SA/HW estimators makes this method a good candidate for implementing in the Control module (responsible for output data collection and for deciding when the precision of final estimates has been reached during runtime) of AKAROA. The results on the performance of SA/HW method, and a comparison of its performance with the other candidates, are given in Chapter 3.

2.1.5 Regenerative Cycles

In methods of regenerative cycles (RC) observations are also grouped into batches, but the batches are of random length, determined by successive instants of time at which the simulated process starts afresh (in the probabilistic sense), i.e. at which its future state transitions do not depend on the past. In the theory of regenerative processes, see for example [SHED87], which gives theoretical support for this method, such instants of time are called regeneration points, and the states of the processes at these points are called regeneration states. The special nature of the process behaviour after each regeneration point — its fresh "re-birth" — causes batches of observations collected during different regenerative cycles (i.e. the time interval between successive regeneration points) to be statistically independent and identically distributed. So are the means of these batches. For example, the regeneration points in the behaviour of simple single-server queuing systems are clearly the time instants at which newly arriving customers find the system empty and idle. From any such moment on, no event from the past influences the future evolution of the system. More examples are given, for example, in [WELC83, p.317] or [SHED87, Sec.2.1]. Note that usually a few, or even infinitely many, different sequences of regeneration points (for different types of regeneration states) can be distinguished in the behaviour of a system.

As a consequence of the identical distributions of output data collected within consecutive regenerative cycles, the problem of initialisation vanishes if a simulation experiment commences from a selected regeneration point. RC were first suggested by Cox and Smith [COXS61, p.136], and then independently developed by Fishman ([FISH73] and [FISH74]), and Crane and Iglehart ([CRAN74], [CRAN75]). Because of the random length of batches, these methods require special estimators, usually in the form of a ratio of two means. Thus, it produces a biased estimator[3] of μ_x (the mean value of the ratio of two variables is approximated by the ratio of their mean values, which generally is not correct). The asymptotic normality of such a ratio estimator is also questionable, even for a relatively large N. The methods of RC eliminate the bias of initialisation but introduce new sources of systematic errors, caused by special forms of estimators. Some efforts have been made to find less biased RC estimators. For example, less biased estimators of μ_x have been proposed in [FISH77] (Tin's estimator), [IGLE75] (the "jackknife" estimator) and [MINH87]. Comparative studies reported in [IGLE75], [IGLE78], [GUNT80] and [LAWK82] show that using the jackknife approach for the mean and variance estimation can significantly improve the accuracy of the estimates, although some question the generality of these results [BRAT83, p.92]. In some reported cases, especially if a small number of regenerative cycles is recorded, the performance of RC appears to be poor indeed, worse than that of NOBM, see [LAWK82], [LAWK84]. To improve the quality of RC estimators Fishman [FISH77] equipped his sequential RC method with a statistical test for normality of the collected data (means over observations collected during consecutive regenerative cycles, grouped into

[3]Nevertheless, it can be shown that this is a consistent estimator , i.e. it tends to μ_x with probability 1, as the number of regenerative cycles $N \to \infty$.

batches). Results presented in [LAWK82] show that this method, although rather more complicated numerically, produces more accurate results in comparison with both a sequential "plain" RC method proposed by Lavenberg and Saver [LAVE77], and a sequential NOBM proposed by Law and Carson [LAWC79]. A sophisticated version of RC was also proposed by Heidelberger and Lewis [HEID81a], who suggested interactive intervention by users in the process of data collection and analysis, to secure better results. Any method of RC requires a regeneration state to be well chosen, to ensure that a sufficient number of output data can be collected for statistical analysis. On the other hand, selecting the most frequently occurring regeneration state does not guarantee the best quality of the estimator. For example, Calvin [CALV88] has shown that such a selection may even result in the estimator with the largest variance. Thus a general criterion for selecting regeneration states still remains an open question. These, and other practical problems with RC, cause their practicality to be questioned, even though theoretically the methods of RC behave very well asymptotically. Any variant of RC offers very an attractive solution to the main "tactical" problems of stochastic simulation, but it requires deeper a priori knowledge of simulated processes' dynamics than Beach Means or SA techniques, making this approach less practical for implementation in a robust automated simulation package.

2.1.6 Uncorrelated Sampling

Uncorrelated sampling (US) can be regarded as a special case of SBM, in which "batches" consisting of single observations, each of s observations apart, are retained, and all other observations are discarded; see [SCMI88] and [SOLO83, p.200]. The distance between consecutive retained observations should be determined to be large enough to make the correlation between them negligible. When this is done, and n_0 observations from the initial transient period are discarded, then K retained observations can be regarded as representing (almost) independent and identically distributed random variables. Thus, the mean μ_x can be estimated by

$$\overline{X}_{US}(K) = \frac{1}{K} \sum_{i=1}^{K-1} x_{n_0+is+1} \tag{2.23}$$

Its confidence interval is

$$\overline{X}_{US}(K) = t_{K-1,1-\alpha/2}\hat{\sigma}_{US}[\overline{X}_{US}(K)] \tag{2.24}$$

where

$$\hat{\sigma}_{US}[\overline{X}_{US}(K)] = \frac{1}{K(K-1)}\sum_{i=1}^{K-1}(x_{n_0+is+1} - \overline{X}_{US}(K))_2 \tag{2.25}$$

and $t_{K-1,1-\alpha/2}$ is the upper (1 - $\alpha/2$) critical point obtained from the t-distribution with (K-1) degrees of freedom. The size s of separating intervals could be selected sequentially, applying the same tests as for determining the batch size in the methods of batch means, although one can expect that the size of intervals used for removing correlations between individual observations will usually be smaller than the batch size required for making batch means uncorrelated. In the example considered in [SOLO83] the separating intervals of length s=25 were selected by applying the Spearman rank correlation test. No results on effectiveness of this method are available, but it is generally accepted that it wastes too many observations. In fact, as shown by Conway [CONW63], the benefit of introducing the separating intervals is doubtful since it increases the variance of estimates. Let us also note that one of the reasons for batching observations is to make them more normally distributed. Thus, US can give quite a poor approximation to the confidence interval, if the analysed process is not a normal one. By these reasons we have considered US as unsuitable for an implementation in the Control Module of AKAROA, our package for automated parallel steady-state simulation and output data analysis.

2.1.7 Standardized Time Series

The methods of standardized time series (STS) rely on the convergence of standardized random processes to a Wiener random process[4] with independent increments, and were originally proposed by Schruben in [SCRU83a], [SCRU85]. It is an application of the theory of dependent random processes

[4]Also known as a Brownian bridge process

[BILL68, Ch. 20 and 21] and its functional central limit theorem, which is a generalisation of the (scalar) central limit theorem. According to this approach, an analysed sequence of observations is first divided into subsequences (batches) of m observations, and then each of them is transformed into its a standard form required by the functional central limit theorem. Next, various functions of the transformed sequence are analysed to construct the confidence interval of $\overline{X}(n)$. The STS methods require that the analysed process is stationary, thus initial observations representing nonstationary warm-up periods should be discarded before the sequence of remaining observations is divided into nonoverlapping batches of a proper size. Schruben [SCRU83a] proposed two original estimators of variance of $\overline{X}(n)$ (called the maximum estimator and the area estimator[5]) and a few of their combinations.

Theoretical studies of these estimators showed that the former estimator is asymptotically superior to the latter one in the sense that, as $m \to \infty$, it produces narrower and more stable confidence intervals, [GOLD84] . On the other hand, it can perform poorly when batches are short. STS estimators have simple numerical forms, despite the sophisticated statistical techniques involved. Their quality can be improved by overlapping batches as in OBM [DAME87]. Recently a new class of weighted area estimators has been introduced [GOLD90]. The search for better STS estimators continues. Unfortunately, no simple rule for selecting the batch size for STS estimators has been proposed. It is only generally claimed that STS requires longer batches than NOBM or OBM, [SONG88] and [SONG88a]. In such a situation implementations of STS estimators in the Control Module of AKAROA, our package for automated parallel steady-state simulation and output data analysis, seem to be impractical at present. The problem of developing a constructive procedure for determining the batch size for STS remains open for future research.

2.1.8 Autoregressive Representation

As all other methods that we have discussed, except IR, the methods based on autoregressive representations (AR) are applied to observations collected

[5] In [SCRU83a] it was called the sum estimator.

during single simulation runs and require that the initial transient observations are discarded before AR is used [FISH71], [FISH73] and [FISH78]. The main feature of AR is that original sequences of correlated observations are transformed into sequences of independent and identically distributed random variables[6].The *autoregressive representation* y_{q+1}, y_{q+2}, ..., y_{q+n} of order q of the sequence x_1, x_2, ..., x_n is defined by the transformation

$$y_i = \sum_{k=1}^{q} c_k(x_{i-k} - \mu_x) \tag{2.26}$$

for $i = q+1$, $q+2$, ..., n. The variance of the mean of such i.i.d. random variables can easily be obtained from Eqn.(4), provided the coefficients q, c_1, c_2, ..., c_q are known. The correct autoregressive order q can be determined by examining the convergence of the distribution of a test statistic to an F distribution[7] or to a χ^2 distribution; see [HANN70, p.336], [FISH78, p.251], or [BRAT83]. Having selected q, the estimates of the coefficients of c_1, c_2, ..., c_q can be found from a set of q linear equations, and one can determine the estimate of the variance of $\overline{X}(n)$ from $\hat{\sigma}^2[\overline{Y}(n-q)]$; see [PAWL90] for details.

The main restriction of AR appears to be the required existence of an autoregressive representation of the simulated process. Results of empirical studies of the method's efficiency published in [FISH71] were not very encouraging. However these results were achieved in short simulation runs. Andrews and Schriber, in their studies of the autoregressive method reported in [ANDR78], [SCRI79] and [SCRI81], observed a significant variability in the average widths of confidence intervals in their simulation experiments. Law and Kelton [LAWK84], after comparative studies of different fixed-size methods of data analysis, found also that AR does not offer better results than other, computationally simpler methods of data analysis. And, in contrast to both BM and SA, the improvement of the final coverage when increasing the number of collected observations was very slow. These results, together with the need for determining the autoregressive order q and solving the sets of

[6]These secondary sequences of data are called autoregressive representations of the original ones.

[7]Also known as the Fisher distribution, or the Snedecor distribution, or the variance-ratio distribution.

equations for determining the coefficients c_1, c_2, ..., c_q, makes AR unattractive for implementations in the Control Module of AKAROA, our package for automated parallel steady-state simulation and output data analysis.

The above survey of existing methods for estimating confidence intervals of steady-state mean values shows the wide diversity among the methods. As mentioned, only three of them, namely the NOBM and OBM methods based on the concept of batch means and the SA/HW method based on spectral data analysis, can be considered as suitable for simulation packages with fully automatic output data analysis and process management in steady-state simulation. Since very little is known about their relative quality, the final choice is an arbitrary one. The results of our comparative studies of these methods are presented in Chapter 3, together with the results characterizing two versions of Independent Replications.

2.2 Problem of initial transient period

Typically, after starting a simulation, the simulated process is initially in a nonstationary phase. Then, if the process is stable, it moves asymptotically toward a steady state. Steady state simulation is aimed to give insight into the behaviour of queuing processes in their steady state. Observations generated during the transient period are realisations of random variables whose distributions differ from those at steady state, i.e. the i-th observation, x_i can be modelled by

$$x_i = \mu + b_i + \epsilon_i \tag{2.27}$$

Hence their use in an estimator of steady state mean can lead to an initialisation bias (b) that is unknown :

$$E[\overline{X}(n)] = \mu + b \tag{2.28}$$

and degrade the reliability of its interval estimate. This raises the question of whether to delete or not to delete initial observations from each replication. Each of these two alternatives has its advocates (see [KELT83], [KELT84] for arguments in support of deletion, and [FISH72], [FISH73], [BLOM70], [TURN77], [WIL78] for arguments against deletion). If deletion is employed, the problem of "how much initial data to delete ?" must be addressed. On the other hand, if it has been decided not to discard data, then it may be helpful to know "whether the estimator bias is significant relative to its standard error ?" when determining whether to collect more observations or to stop. Alternatively, for a fixed given simulation length, it would be useful to have at least an indication of the extent to which the estimate may be biased.

Various tests have been developed for estimating the "end" of the transient period, beyond which data collected would contain an insignificant bias. During simulation, these methods decide whether the system has reached steady state by testing whether its output satisfies a property that is thought to be a characteristic of the simulated process in its steady state [CONW63], [TOCH63], [FISH73b], [GAFA78], [ROTH85], [MORS55], [SCHR82], [SCHR83]. Typically, these tests are founded on some assumptions on the behaviour of b_i as a function of i, e.g., that the function is of quadratic or exponential form, and/or that it is a monotonic function. Other requirements might be that when the test is carried out, the simulated has progresses beyond the point where the transient is significant, i.e. only an initial subsequence x_1, x_2, ... , x_j of the observations collected, x_1, x_2, ... , x_n, $j < n$, would have been collected during the transient period. In fact, it was assumed in [JACK92] that "if we have absolutely no knowledge of the time-scale of the transient then no possible bias detection could work since the total data at hand may conceivably be only an infinitesimal section of the transient".

Here, we identify three approaches for dealing with the problem of initial transient bias. The first approach is to estimate the length of the initial transient period, and discard data collected during that period. The argument in support of this approach is that it should reduce the bias of the estimator, and hence improve its quality in terms of its coverage. The second approach is to ignore the presence of an initial transient period, and assume that all data were collected from the process in its steady state. The argument in support of this approach is that, by not discarding any observations, more observations are available for producing the estimate, so the variance of the estimator, and hence its relative precision should be improved. However, observations collected during the initial transient period neither belong to a stationary sequence nor characterise steady-state behaviour of the simulated process. Neglecting the existence of initial transient periods can lead to significant bias in steady-state estimates of analysed performance measures. The third approach, developed by us, is to directly consider the bias in the estimator, and also refrain from discarding any observations. According to this approach, either the interval estimate for the mean is adjusted to accounted for the estimated bias, or the length of each replication is increased until it has being determined that the biased caused by observations collected during the transient period is small, relative to the variance of the esitmator [YAU96]. As mentioned, this approach is represented by the method of Polarised Independent Replications. The Polarised Independent Replications is firstly described, then methods based on the discarding of initial observations will be discussed.

2.2.1 Polarised Independent Replications : A Method Based on Assessment of the Bias in an Estimator

Polarised Independent Replications (PolarIR), attempts to address both the problems of correlated data, and the problem of inaccurate interval estimates caused by initialisation bias. PolarIR is intended for estimating mean values such as mean queue length, throughput, or mean delay, either in open queuing systems where all queues have finite capacity, or in closed queuing networks with finite or infinite queues.

Instead of launching all replications from the same initial state, the method of PolarIR launches half of the replications initialised to the empty-and-idle (-ve) state, and half of them from the full-and-busy (+ve) state, when estimating a parameter such as mean queue length, throughput, or waiting time. At various checkpoints, the average of the observations from all replications are collected to form a point estimate of the mean of interest (e.g. mean throughput). If the estimate has reached the required level of precision, and if the bias of this estimate is statistically insignificant compared to the standard deviation of the estimator (standard error), then the simulation could be stopped.

The idea of PolarIR is as follows. Like IR, k_r replications are generated with m observations collected during each of them, however, $k_{r/2}$ of then are started with the system initialised to the empty-and-idle (-ve) state giving

$k_{r/2}$ sequences of data, $(x^-_{11}, x^-_{12}, ..., x^-_{1m})$, $(x^-_{21}, x^-_{22}, ..., x^-_{2m})$, ..., $(x^-_{k_{r/2}1}, x^-_{k_{r/2}2}, ..., x^-_{k_{r/2}m})$; and $k_{r/2}$ of then are started with the system initialised to the full-and-busy (+ve) state giving $k_{r/2}$ sequences of data, $(x^+_{11}, x^+_{12}, ..., x^+_{1m})$, $(x^+_{21}, x^+_{22}, ..., x^+_{2m})$, ..., $(x^+_{k_{r/2}1}, x^+_{k_{r/2}2}, ..., x^+_{k_{r/2}m})$;

No initial observations are deleted. The means of replications started with the system initialised to the -ve state are then calculated as :

$$\overline{Y}^-_i = \overline{X}_i(m) = \frac{1}{m}\sum_{j=1}^{m} x^-_{ij} \quad i = 1, 2, ..., k_{r/2} \tag{2.29}$$

The secondary data, Y^-_1, Y^-_2, ... , $Y^-_{k_{r/2}}$, can be considered as realisations of i.i.d. random variables that can be used to get the point and interval estimator of μ_x by substituting n by k_r, x_i by $\overline{Y}^-$, and $\overline{X}_n$ by

$$\overline{X}^-_{IR} = \frac{1}{k_{r/2}}\sum_{i=1}^{k_{r/2}} \overline{Y}^-_i \tag{2.30}$$

Likewise the means of replications started with the system initialised to the +ve state are calculated as :

$$\overline{Y}^+_i = \overline{X}_i(m) = \frac{1}{m}\sum_{j=1}^{m} x^+_{ij} \quad i = 1, 2, ..., k_{r/2} \tag{2.31}$$

The secondary data, Y^+_1, Y^+_2, ... , $Y^+_{k_{r/2}}$, can be considered as realisations of i.i.d. random variables that can be used to get the point and interval estimator of μ_x by substituting n by k_r, x_i by $\overline{Y}^+$, and $\overline{X}_n$ by

$$\overline{X}^+_{IR} = \frac{1}{k_{r/2}}\sum_{i=1}^{k_{r/2}} \overline{Y}^+_i \tag{2.32}$$

Using PolarIR, the final point estimator of μ_x is

$$\bar{\bar{X}}_{IR} = \frac{\overline{X}^-_{IR} + \overline{X}^+_{IR}}{2} \tag{2.33}$$

Assuming that the bias of $\overline{X}_{IR}^{+}$ is positive (since the replications were initialised to the full-and-busy state), and that the bias of $\overline{X}_{IR}^{-}$ is less than or equal to zero (since the replications were initialised to the empty-and-idle state), then the maximum value of the bias in $\bar{\bar{X}}_{IR}$ can be estimated as

$$B_{max}(\bar{\bar{X}}_{IR}) = \overline{X}_{IR}^{+} - \overline{X}_{IR}^{-} \tag{2.34}$$

The test of whether $B_{max}(\bar{\bar{X}}_{IR})$ is a significant component of the MSE of the PolarIR estimator is equivalent to testing

$H_o : [B_{max}(\bar{\bar{X}}_{IR})]^2/(\hat{\sigma}_{US}[\overline{X}_{US}(K)] = 0$

versus

$H_A : [B_{max}(\bar{\bar{X}}_{IR})]^2/\hat{\sigma}_{US}[\overline{X}_{US}(K)] \neq 0$

Recall that $\overline{X}_{IR}^{+}$ and $\overline{X}_{IR}^{-}$ are the means from replications launched from +ve and -ve replications respectively. Thus we cannot automatically assume that their variances are equal. However power of the test depends on whether one can assume var($\overline{X}_{IR}^{+}$) = var($\overline{X}_{IR}^{-}$). Thus the test procedure involves firstly testing whether the variances of $\overline{X}_{IR}^{-}$ and $\overline{X}_{IR}^{+}$ are equal. If their variances were concluded to be equal, then the F ratio test can be applied. Otherwise, H_o would be tested using the using the Behrens-Fisher test.

Polarised Independent Replications has the following advantages:

- Assumptions on the behaviour of the transient bias required for PolarIR are quite relaxed. Namely, the mean bias of replications launched from one +ve and -ve states must have opposite signs, or are zero. This assumption is quite different to the assumptions required by other transient detection tests (which typically require quadratic, exponential, or monotonic behaviour). Thus PolarIR may be used when other methods are not applicable.

- With PolarIR, a decision on "whether the simulation has run long enough for the bias to be insignificant ?" would be based on well established statistical inference procedures, instead of a heuristic.

- By launching replications from +ve or -ve states instead of the same initial state, there is partial cancellation of bias in the final estimator (one over estimates, and one underestimates).

- NO initial data deletion, yet the effect on the quality of the estimate due to initialisation bias would be known to be insignificant.

However it should be cautioned that this method has not been evaluated empirically yet. The main drawback of PolarIR is that it is not applicable to all systems. For example, the PolarIR method cannot be applied to open queuing network with queues with infinite capacity.

2.2.2 Method Based on Discarding Observations Gathered During the Initial Transient Period

Since observations gathered during the initial transient periods do not characterise the steady state, a natural idea is to discard all such observations before further analysis. This requires an estimation of the effective length of the initial transient period. Ignoring the existence of this period can lead to a significant bias of the final results. Although, the removal of any observations increases the variance of estimates, which in turn can increase the value of the mean-square error (cf. [DONN81], [TURN77], [WILS78]), the bias of estimates is generally regarded as the more important factor when considering the quality of estimates, and it justifies our decision of implementing an efficient procedure for detecting the length of initial transient periods in the Control Module of AKAROA.

Basic problems related with the existence of initial transient periods and the detection of their lengths are discussed in [PAWL90]. There have been two methods for automatic detection of the length of initial transient period proposed: one proposed by Pawlikowski [PAWL90], and another one proposed by Jackway and deSilva [JACK92]. The later is a modified version of the former. No comparative studies of these two methodologies have been done yet. Basing on our own experience, the former has been selected to be implemented in the Control module of AKAROA. It follows the following steps:

A rough, first approximation of the number $n_d(0)$ of initial observations that should be discarded is obtained applying one of heuristic rules of thumb surveyed in [PAWL90]; namely the rule R5 of that survey:

> the initial transient period is over after no observations if the series of observations x_1, x_2, ..., x_{n_0} crosses the mean $\overline{X}_{n_0}$ k times
> .

This rule is sensitive to the value of k, see [GAFA78]. Too large a value will usually lead to an overestimated value of n_0 regardless of system utilization, while too small a value can result in an underestimated n_0 in more heavily loaded systems. Results of our studies confirmed the justness of selecting k=25, recommended in [GAFA78].

Following the first rough selection of the truncation point for initial data, the length of the initial transient period is more precisely determined sequentially, by applying one of statistical tests proposed in [SCRU82] and [SCRU83] for testing (un)stationarity of collected observations. These tests are based on the high sensitivity of the sequence of partial sums

$$S_k = \overline{X}(n) - \overline{X}(k)\ , \tag{2.35}$$

(k= 0, 1, 2,..., and S_0=S_n=0) to the presence of initialization bias in $\overline{X}_n$; where $\overline{X}_n$ and $\overline{X}_k$ are means over n and k first observations, respectively, see Eqn. 2.1. Following this phenomenon, the tests analyse the convergence of a standardized sequence {T(t)}, 0≤ t ≤1, to the Brownian bridge process with zero mean and variance equal 1. The sequence {T(t)} is the standardized sequence of the partial sums S_k, namely

$$T(t) = \lfloor nt \rfloor S_{\lfloor nt \rfloor} \frac{1}{\sqrt{x}\hat{\sigma}[\overline{X}(n)]} \tag{2.36}$$

for 0< t ≤1 (where $\lfloor x \rfloor$ denotes the greatest integer not greater than x, and T(0) = 0. Rejection or acceptance of the hypothesis that a given subsequence of observations is stationary, or equivalently, that the initial transient period is not included in the observations, depends on the probability characterizing the scalar value calculated from the considered sequence. Despite the sophisticated theory hidden behind these tests they appear to be quite simple numerically, and can be applied to a wide class of simulated processes. A practical problem faced when implementing one of these tests is that they require a priori knowledge of the steady-state variance $\sigma[\overline{X}(n)]$ of the simulated process. As proposed in [PAWL90], it has been solved in AKAROA by estimating this variance from a sequence of observations collected some time after the assumed supposed-to-be truncation point, assuming that then the process is at least closer to its steady state.

A logical description of the sequential version of the stationarity test applied in AKAROA is given in below in its pseudo-code form. It should be noted that the actual stationarity test algorithm implemented in the DetectInitialTransient object (C++ class) of AKAROA has been inverted from the description given in Fig. 2.1 and reassembled, so that the "data generation process", and the "detect initial transient" are separated, and mutually hidden. Thus the "data generation process" interacts with the "detect initial transient" process only through a well defined interface, and the implementation of either can be changed without affecting the other, so long as they perform their respective tasks.

Figure 2.1: A logical description of the sequential version of the stationarity test applied in AKAROA. Note that to separate the "data generation process", from the "detect initial transient", the actual stationarity test algorithm implemented in the DetectInitialTransient object (C++ class) of AKAROA has been inverted from the procedure specified here

The structure of the objects for steady state point and interval estimation based on the NOBM, OBM, IR, and SA/HW methodologies selected in this Chapter as candidates for implementation in the Control module of AKAROA, and the object for detecting when the effects of the initial transient are insignificant, as well as a comparison of their performance, will be considered in Chapter 3.

Chapter 3

Comparative Studies

Following the survey of techniques for precision control of steady-state estimates and initial transient detection in previous chapter, we have selected the NOBM, OBM and SA/HW method as the candidate methods for implementation in the control module of AKAROA, our package for automatic generation of parallel simulators and the control of steady-state simulation processes and output data analysis in parallel stochastic simulation. These three methods are also referred to below as BatchMeans, OBatchMeans and Spectral, respectively. Additionally, the results characterizing three other methods of sequential estimation of the precision of steady-state estimates have been presented. Namely:

- two versions of Independent Replications, referred to as "Indep.Repl" and "Indep.Repl 3+", where the former can be stopped with as little as three replications collected, while the latter required more replications to be finished, protecting against stopping simulation too soon, with too few data collected, and
- a naive method referred to as "classical", that ignores correlated nature of collected data and estimates widths of confidence intervals using Eqn. 2.4.

3.1 Methodology

3.1.1 Performance Metrics

Performance of the methods analysed is compared here in the sense of :

- the coverage, i.e. their experimental confidence level, as will be described later,
- the required simulation run length, measured by the number of observations needed for stopping the simulation with the required level of precision, and
- the final precision of the results at the end of simulation.

Traditionally, the quality of various approximations of interval estimation methods has been analysed experimentally. Let us note that in an ideal case the final confidence interval would contain the true value of the estimated parameter with only the probability 1-α, where α is the required level of confidence. Equivalently, if an experiment is repeated many times, then the interval estimators produced from them would contain the true value in (1-α)% of cases. Thus the generally accepted empirical measure of the robustness of the interval estimators is the *coverage of confidence intervals*, C, defined as the frequency with which the intervals $(\overline{X}(n)-\Delta_x, \overline{X}(n)+\Delta_x)$ contains the true mean μ_x, at a given confidence level (1-α), $0\leq \alpha \leq 1$, see Fig. 3.1, where Δ_x is as defined in Eqn. 2.3.

Since the value of μ_x has to be known, coverage analysis has to be limited to systems with theoretically well known behaviour. An analysed interval estimator must be applied to a sufficient number of independently[1] repeated simulation experiments to get a good estimate of its coverage. This practically means repeating each experiment *at least* 200 times per working point. As a result the coverage estimate $\hat{C}$ s.t.

[1] using mutually independent sequences of pseudo-random numbers

Figure 3.1: Valid and invalid confidence intervals for μ_x in coverage analysis

$$\hat{C} = \frac{\textit{number of experiments in which the interval estimate contains the true mean}}{\textit{total number of independent experiments}} \tag{3.1}$$

can be regarded as approximately normally distributed, and one can determine the confidence interval of C by applying the standard formula for large sample confidence interval for proportion

$$\left(\hat{C} - z_{1-\alpha/2}\sqrt{\frac{\hat{C}(1-\hat{C})}{n_{\hat{C}}}}, \hat{C} + z_{1-\alpha/2}\sqrt{\frac{\hat{C}(1-\hat{C})}{n_{\hat{C}}}}\right) \tag{3.2}$$

where $z_{1-\alpha/2}$ is the (1-α/2) quantile of the standard normal distribution, and n_c is the number of experiments in coverage analysis. An output analysis method for producing interval estimates is considered as producing valid 1001-α confidence intervals if the upper bound of the confidence interval of its coverage is at least (1-α). Otherwise the confidence intervals for the estimated parameter should be regarded as invalid, and the method of estimation should be regarded as inaccurate.

Since the coverage analysis requires that the exact characteristics of the simulated processes are known, the M/M/1/∞ queuing system (i.e. a queuing system with Poisson stream of arriving customers, exponentially distributed service times, one server, and infinite buffer capacity) was assumed as the reference system for studying performance of the methods. Let us note that from the viewpoint of the system's dynamics, the $M/M/1/\infty$ queue can be considered as the worst case system, and by this reason, is commonly accepted as a reference model in experimental studies of simulation techniques. We analysed the mean-delay of customers arriving at $M/M/1/\infty$, and all estimates were required to have the relative precision of 5% (or less), at the 0.95 confidence level. Notice that for the purpose of coverage analysis, it does not matter whether we estimate the mean delay, or the mean waiting time, even though in practical situations a simple variance reduction technique is to estimate only the mean waiting time by simulation, and from that estimate the mean delay. To ensure statistical representativeness all reported results are averages over 2000 independent replications, repeated at each considered working point of the $M/M/1/\infty$ queue.

3.1.2 Implementation

As mentioned, in contrast to the algorithms specified for some of the above methods given in [PAWL90], we decided to separate the simulation task into two, almost autonomous processes :

1. data generation process,
2. detect initial transient, and the steady state estimation

which cooperate via well defined interfaces. The task of data generation can be performed by any user simulator written in C or C++. The task of this "un-instrumented" simulator is to solely model the system of interest. Thus the programmer of the simulator is freed from all statistical analysis procedures, data processing and storage, and runlength control.

The tasks of detecting the (approximate) end of the initial transient period, and of steady state estimation is performed by a separate, precision control object. As mentioned, five precision control objects were implemented by us in order to conduct a comparison of the candidate methods, namely the Batchmeans, OBatchmeans, Spectralanalysis, IReplication, and IReplication3 objects were developed for implementing the NOBM, OBM, SA/HW, Indep.Rep, Indep.Rep3+ methods of point and interval estimation and run-length control.

Thus the "data generation process" interacts with the "precision control process" only through a well defined interface, and the implementation of either can be changed without affecting the other, so long as they perform their respective tasks. This also allows the user to focus only on developing and testing a simulator of the system of interest, without being burdened with the functions of data collection, batching and/or consolidation, nor with the complexities of initial transient detection, nor the of statistical analysis data collected during steady state and runlength control. Afterwards the user completed an (un-instrumented) simulator, any one of the candidate precision control objects can be chosen to perform these functions by simply attach it to the simulator.

The execution path followed by a typical simulator is depicted in Fig. 3.2. During the execution of the simulator, the $processnewobs(new\_observation)$ member function of the precision control object is invoked whenever a new observation is generated. If the call to $processnewobs(new\_observation)$ returns STOP then sufficient observations had been collected and the simulation could be stopped, and results written. Otherwise the simulation continues generating observations, feeding them to the precision control object. This call by the basic simulator to the $processnewobs(new\_observation)$ member function of the precision control object is almost the only interaction between the simulator (developed by the user) and the precision control object.

For better memory utilization, each precision control object was implemented dynamically, i.e. composed of two (sub)objects, each of which being created only when its services are required. The first constituent object is for testing the end of transient phase, and the other is for steady-state precision control. For a fair comparison, we used the same method for detecting when the simulated process has progressed to the stage where the effect of the initial transient can be considered as insignificant. This method was implemented as a C++ object called DetectInitialTransient (refer to section 2.2 for a logical description of this method). The transient object should be created only when the first observation has been obtained, and does not occupy resources before then. This transient object is disposed after the steady-state

Figure 3.2: Flowchart of the execution path of a typical simulator, and its interaction with the precision control object

stage has been detected, releasing the resources it occupied (its instruction and data space) to the system.

In a similar manner, the steady-state output analysis and precision control object is created only after steady state is reached, and is disposed of when the desired precision of a given parameter is obtained. Accordingly, when several parameters are being estimated simultaneously, the control object would create a set of such pairs of objects (transient and steady-state one)

, one pair for each parameter when needed, and dispose them when a given estimate is determined with a sufficient precision. All these operations should be fully transparent for users.

The execution path followed by a typical precision control object[2] is depicted in Fig. 3.3. During the execution of the simulator, the $processnewobs(new_observation)$ member function of the precision control object is invoked whenever a new observation is generated. This function takes one of two conditional actions :

1. If the process was last known to be in its transient phase, then it invokes the $processnewobs(new_observation)$ member function of the DetectInitialTransient object. If the function determined that the process has reached steady state, it returns ETRP (end of transient period detected). Otherwise it returns ITRP.

2. If the process was known to have reached steady state, then the $processnewobs(new_observation)$ member function of the steady state analysis object is invoked. As mentioned one type of steady state analysis object was implemented for each of the methods evaluated. The analysis object is responsible for collecting steady state observations of the parameter being estimated, and for their analysis at specific checkpoints to produce a point and interval estimate of the parameter. If, having reached a checkpoint, its determines that the point estimate has reached the required level of precision, then it would return EPOK to the $processnewobs(new_observation)$ member function call, which indicates that the estimate has achieved the required level of precision. Otherwise, the function returns CONT which indicates that more observations are required.

If ETRP was returned, then it invokes the $processnewobs(new_observation)$ member function of the steady state analysis object. If ITRP was returned, it

[2]The exception is if an IR method was used, in which case the simulation processes are created by a parent precision control process. Estimates from each replication are returned to the parent before the replication terminates. The control process then forms a new estimate, and repeats this procedure until the estimate first achieves or exceeds the required precision.

returns CONT to the simulator, to indicate that it should continue producing observations.

If CONT was returned (from the $processnewobs(new_observation)$ member function of the steady state analysis object), then it is returned to the invokee (i.e. the simulator). If EPOK was returned, then the $processnewobs(new_observation)$ member function of the precision control object checks if all parameters being estimated during the simulation run has achieved the required level of precision. If so, then it returns STOP to the simulator, otherwise CONT is returned.

If the call made by the simulator to $processnewobs(new\_observation)$ returned STOP then sufficient observations had been collected and the simulation could be stopped, and results written. Otherwise the simulation continues generating observations, feeding them to the precision control object, see Fig. 3.2.

Collection, batching, and consolidation of data items must be performed by each type of control object, in addition to statistical analysis. The types of operations, the unknown number of data items to be stored, and the nature of the execution environment were the main factors influencing our choice of data structures and access functions for their implementation. In the object responsible for detecting initial transient a doubly linked list was implemented for storing observations, since the number of observations required for testing is here unknown in advance, and the main operations involve a series of references either at the beginning or near the end of the sequence of collected observations. With a linked list, memory is allocated only when it is required and it can be freed to the system when an observation is no longer needed.

The number of observations collected by NOBM or OBM objects grows in time during a simulation run, and data items are usually accessed sequentially. Thus, in this case batch means may be stored in a singularly linked list. The number of consolidated data items used by the Adam's test when testing for the level of correlation between batches (see [ADAM83] [PAWL90]) is bounded and relatively small, so they were stored in an ordinary one-dimension array. A nice property of SA/HW is that the storage needed for data collected during steady state is bounded and limited to 200, due to the use of data compression. This property, combined with the need

Figure 3.3: Flowchart of the execution path of a precision control object, and its interaction with the basic simulator

to access most of the collected data during every checkpoint lead us to use a one-dimensional array.

3.2 Results

Fig. 3.4 shows the coverage as a function of the normalized load ρ for all considered methods of controlling the simulation length. As we see, correlations among observations have to be taken into account, since the coverage of "classic" never reaches an acceptable level, and decreases quickly to zero with increasing traffic level !! On the other hand, all other methods considered, that take into account existing correlations perform similarly in low traffic level, but both versions of IR that we considered give very poor coverage when the system becomes heavier loaded. All three methods that we selected as candidates for being implemented in AKAROA offers similar coverage in high traffic region, although the coverage of SA/HW is the highest one, while Nonoverlapping Batch Means gives worse coverage than Overlapping Batch Means. The average final precision of estimates, i.e. the precision at the simulation stopping points, obtained under the methods considered is shown in Fig. 3.5. As we see, the final precision of estimates under SA/HW is practically constant and independent from the traffic load. It is caused by the fact that the distance between checkpoints under SA/HW increases geometrically, causing more significant changes in precision of estimates with the time of simulation; see [PAW90] for details. Despite that in all other methods considered the step between consecutive checkpoints is kept constant, the final precision of estimates is getting closer to the limit value ϵ_{max}, which suggest that the convergence rate of precision to the required level slows down with the time of simulation. It seems to be a consequence of the fact that the variance of estimators is inversely proportional to the number of collected observations.

Fig. 3.6 and 3.7 show the average simulation run lengths, measured by the numbers of observations needed for getting estimates with the required precision. While SA/HW uses, on the average, the shortest simulation run lengths, in heavier loaded systems it requires longer simulation runs than other methods. Especially of interest is the observation that in heavy loaded systems SA/HW can require longer simulation run lengths than what is theoretically needed to obtain an estimate at the ϵ=$\pm$0.05 level of relative precision. This can be due to two factors. Firstly the theoretical value states the number of *steady state* observations required, whereas the figure for the average runlength required by SA/HW is the total runlength (including the

Figure 3.4: Coverage of the final confidence intervals of the mean customer delay in M/M/1/∞ queue

observations collected during the transient phase which were deleted). Even if we subtract the average number of discarded observations, the runlength required by SA/HW may still exceed the theoretical length because the estimates produced by the SA/HW on average exceed the required level of relative precision, see Fig. 3.5.

For a fair comparison, the problem of detecting the end of the initial transient period was addressed using the same methods in all simulation experiments, regardless of the method used for steady state analysis. Thus the NOBM, OBM, SA/HW, IR, IR3+ precision control objects all estimated the length of the initial transient phase of the analysed process using the same DetectInitialTransient (sub) object. DetectInitialTransient applies the tests specified in Chapter 2; namely it firstly use Gafarian's Heuristic with k=25 crossings to obtain a rough estimate of the end of the transient phase, followed by repeated applications of Schruben's test until steady state conditions were

Figure 3.5: The average relative precision of the final steady-state estimates of the mean customer delay in M/M/1/∞ queue

detected. In all cases, the observtions collected during the transient period were discarded. Fig 3.8 shows the mean number of observations collected during initial transient periods during our simulation of the M/M/1/∞ reference system, as determined using the DetectInitialTransient object, for different levels of the system's load.

3.3 Conclusions

Following an examination of all the results we obtained and reported here, and having recognised the generally superior coverage of the SA/HW method, especially in heavy loaded systems, and in view of the fact that the data space required by SA/HW is bounded, and typically less than that of OBM and NOBM due to the use of data compression, we have decided to choose

Figure 3.6: The average simulation run length for estimating the mean customer delay in $M/M/1/\infty$ queue, with the relative precision ≤ 5 %, at 0.95 confidence level.

SA/HW as the basis of the precision inference method to be developed for implementation in the Control module of AKAROA, our package for automatic generation of parallel simulators and the control of steady-state simulation processes and output data analysis in parallel stochastic simulation.

Approaches to employing parallel processing in stochastic simulation, and the methodologies that has been previously considered for precision control of the esitimators in the parallel simulation context, and the method based on SA/HW that has been implemented in AKAROA, will be presented in the next chapter. The design goals of AKAROA, and its fuctionalities from the *user's* point of view as specified by its user programming and runtime interface will be considered in Chapter 5. The architecture, implementation, and the results of initial performance studies of AKAROA are reported in Chapter 6.

Figure 3.7: The average simulation run length for estimating the mean customer delay in $M/M/1/\infty$ queue, with the relative precision ≤ 5 %, at 0.95 confidence level.

Figure 3.8: The mean length of the initial transient period (measured by the number of observations generated during the initial transient) as determined by the DetectInitialTransient object at 0.95 confidence level during simulations of the M/M/1/∞ queue.

Chapter 4

Applications of Concurrent Processing in Stochastic Simulation

Simulations in Engineering and Computer Science are often computationally intensive tasks, requiring long runs in order to obtain results at a desired level of precision. Excessive runtimes hinders simulator development and validation, and may inhibit the performing of production runs. This problem is often magnified, as many runs may be needed to obtain an understanding of the simulated system, and more runs may be required to answer questions arising from the results. In fact often initial runs of a simulation model generate more questions than answers and the flow of the study is changed.

Fortunately multiprocessor and distributed systems do offer high potential processing power, many times that available with a uniprocessor. The challenge then, is to develop simulation methodology that can exploit the enormous power and economic advantage of multiprocessors and multicomputer networks to speedup simulation runs. In general there are two classes of parallel stochastic simulation techniques. Specifically, in *concurrent stochastic simulation*, meant as any stochastic simulation that applies the ideas of concurrent processing, one can execute either:

1. *single replication in parallel* (SRIP), where many processors cooperate

in running a single replication of a simulated system, or

2. multiple replications in parallel (MRIP), with processors engaged into running their own, independent replications of the simulated system, but cooperating with central analysers of simulation output data.

Research in parallel simulation has been almost entirely focused on SRIP, as on an interesting problem of parallel processing, while MRIP has been considered only in a few publications despite that this is an attractive scenario from the point of view of statistical methodology.

The execution of the simulation model in SRIP scenario can be achieved in a synchronous or asynchronous way, thus for our purposes we further subdivide SRIP simulations into *synchronous SRIP* (S.SRIP) and *asynchronous SRIP* (A.SRIP). In S.SRIP the basic computational activities are executed synchronously with respect to the simulation clock, i.e. the execution of activities associated with the next event in discrete event simulation cannot be initiated before activities associated with the previous event (performed possibly in parallel) has been finalised. In contrast, in A.SRIP, a given simulation model is partitioned into interdependent parts that are executed in parallel. Thus the basic difference between these two categories of SRIP scenarios is in the level at which concurrent processing is applied.

In S.SRIP, support functions demanded by discrete event simulation such as random variate generation, statistical analysis, and input/output are distributed onto different workstations or processors. Viewing discrete event simulation as a combination of model evaluation and support functions, S.SRIP attempts to speedup a simulation by allowing some tasks in the 'support functions' subset of activities to be performed concurrently with 'model evaluation' tasks. Conversely, with A.SRIP, the simulation model of a system is partitioned into interdependent parts that are executed in parallel, to shorten the total time of simulation. The 'system model evaluation' task is decomposed and distributed.

On the other hand, within the class of MRIP techniques, multiple replications of the simulation model are executed on several processors. Thus the entire model would be replicated for execution on several processors simultaneously.

Each class of parallel simulation methods has its own merit in terms of exploiting the potential of multi-computer or multiprocessor systems. However, each of them has its own weakness, thus the capabilities and characteristics of all three classes of methods must be thoroughly examined to put a newly developed parallel simulation methodology into proper perspective. Full comparison should include such aspects as their transparency (from the user's point of view), scalability, degree of processor utilisation, statistical quality of estimators used in each scenario, and fault tolerance. Various parallel simulation approaches of each of the two classes, and an assessment of them from the perspective of sequential stochastic simulation, as well as programming environments developed for their support, are reviewed in this chapter.

4.1 Single Replication in Parallel

As mentioned concurrent stochastic simulation within the single replication in parallel (SRIP) class can be subdivided into two different categories :

1. *synchronous* single replication in parallel (S.SRIP), and
2. *asynchronous* single replication in parallel (A.SRIP).

4.1.1 Synchronous Single Replication in Parallel

Simulations within the synchronous SRIP (S.SRIP) category attempts to speedup discrete event simulations by executing support functions provided by a simulation language such as random variate generation, event list manipulation, and statistical analysis of output data, on separate processors [WYAT83],[KRIS85]. All model code written by the user is executed sequentially on a single processor, therefore the semantics of a program of a model for S.SRIP is equivalent to that if it was executed on a uniprocessor. For instance, Krishnamurthi et al reports their implementation of S.SRIP for the GASPIV simulation language on a shared memory, time-shared bus connected multiprocessor. S.SRIP's major strength is that it can easily be implemented transparently, freeing the user from the complexities parallel processing. Another beneficial consequence of sequential model evaluation is that existing precision control mechanisms, developed for simulators executed sequentially on single processors, can be safely applied.

Nevertheless model evaluation often takes a dominant share of simulation runtime [BRIN88]. In [BRIN88] Briner and Jack found that for an electrical timing simulator executing average simulations, almost 70% of the total runtime was spent on model evaluation. Consequently when model evaluation is the bottle-neck in running a simulation, introducing parallelism in performing support functions may not yield any appreciable speedup [BURK90]. Another drawback of S.SRIP is that the degree of parallelism in performing support functions is limited, thus restricting the number of processors that can be productively employed. Lastly the fine granularity of decomposition of support functions needed to enable their parallel execution necessitates frequent communication among subprocesses. Apart from consuming processor power, interprocess communication (IPC) limits the multiprocessor architectures that can be used for parallel simulation. For concreteness, in [BOWE94] we found that as communication between processes on different processors increased in time-shared bus and multiple-bus multiprocessor systems, first the bus becomes a bottleneck, increasing the amount of time that processors are blocked, waiting for access to a common memory module (in shared memory architectures, processes communicate through common memory). Yet if the number of shared buses were increased, memory contention (the delay as processors queue for access to common memory module(s)) forms another bottleneck, again limiting effective processing power.

4.1.2 Asynchronous Single Replication in Parallel

Asynchronous SRIP methods attempts to speedup discrete event simulations by executing portions of the simulation model in parallel, and has been the focus of tremendous research efforts in the past decade [MISR86], [MISR86a], [CHAND83], [NICO88a], [FUJI90], [PAYN91]. In A.SRIP the programmer must break the system being modelled into a number of physical processes and fully characterise the rules for their interaction. To illustrate, in a DQDB network simulation model, the physical processes might initially be network nodes that interacts by transmitting requests and segments over the slotted buses. Further decomposing, each node process would yield four (sub) physical processes, one pair each for accessing upstream and downstream buses, and for each direction, a process for handling request slots on a bus and one for processing data bus slots. Interaction rules are derived from the DQDB protocol.

The A.SRIP simulator is constructed by designing a set of logical processes, one per physical process, which simulate the associated physical process. Thus shared objects of sequential discrete event simulation, i.e.. the global clock and event list, are discarded. Specifically no shared variables are used in A.SRIP. Each logical process maintains a local clock that denotes (its own view of) simulated time, and a portion of the state corresponding to the physical process it models. Interaction between processes is done through passing of *messages* between processes, and since each process executes in its own local simulated time, messages must be stamped with the time of the sending process. For example a physical process P1 which causes the scheduling of an event at time t in process P2 is simulated by the logical process of P1 sending a message with the time stamp t to the logical process of P2. The major difficulty with A.SRIP is that logical processes progress at different simulated times, and must therefore somehow be synchronised sufficiently so that causality errors that may result are averted or corrected. A.SRIP mechanisms can be broadly classified into two groups:

1. *conservative methods*, that strictly avoid the possibility of any causality error ever occurring, and

2. *optimistic methods* for distributed scheduling of events, in which causality errors can occur.

Conservative A.SRIP Simulation Methods

Conservative mechanisms ensures that before an event message received by a process is processed, every prior event has already been executed by a processor in the system. Conservative A.SRIP mechanisms were first proposed independently by K. M. Chandy and by R. E. Bryant. Chandy and Misra [CHAN81], Chandy et al [CHAN79], and Bryant [BRYA77] reports a mechanism for conservative A.SRIP, that is the basis of many later variants.

In the original Chandy-Misra-Bryant method (described in [CHAN79] and [BRYA77]), a clock is associated with each input of a logical process and contains the time of the first outstanding message on that input, or, if the input is empty, the time stamp of the last message received over it. Each logical process repeatedly selects the input with the smallest clock value and, if it is not empty, receives the first message on that channel. If it is empty, the process blocks on every channel that has the smallest clock value. Although their protocol ensures that no causality error will ever occur, it may result in deadlock [GROS89],[CHAN83],[PAYN91], where each process is blocked, waiting on the incoming link containing the smallest clock value. Several mechanisms have been proposed to solve this problem. In the original method, *null messages* are used to avoid deadlock situations. A null message with the (sender's) timestamp equal to t_{null} is a indication by the sender that it will not send a message carrying a timestamp smaller than t_{null} to the receiving process. Chandy and Misra in [CHAN79] showed that this null message mechanism avoids deadlock as long as one does not have any cycles in which the collective timestamp increment of a message traversing the cycle could be zero.

An extension to the null message method is to send null messages on a demand basis rather than after every event [SU89], [MISA86a]. This scheme helps reduce the amount of null message traffic, although a longer delay may be required to receive null messages because a null message request message must be transmitted first.

An alternative to allowing, detecting and recovering from deadlock using null messages, is to enforce synchronous process execution by iteratively determining which events are safe to process, and then processing them in parallel [LUBA89]. Barrier synchronisation is used to keep iterations from interfering with each other.

Still other improvements to the basic Chandy-Misra method have been investigated. Nicol [NICO88a] studied a variant which exploits knowledge known about a specific model to predict what or will not happen in the simulated future. Called *lookahead*, the ability to peek into the future based on model dependent knowledge is used to determine which events are safe to process (i.e. whose execution now will not result a causality error). Applied to the deadlock avoidance approach, it can be used to determine the timestamp that is assigned to null messages, which may be equal to or larger than that if lookahead was not applied.

Optimistic A.SRIP Simulation Methods

Optimistic mechanisms allows event carrying messages to be processed as soon as they arrive at any given logical process. Since logical processes advance asynchronously, each progressing in its own local simulated time, causality errors may occur. A causality error is possible whenever an event arrives which has a smaller timestamp than an event already executed. The event corresponding to the message with the smaller timestamp is called a *straggler*.

In the classical Time Warp mechanism [GAFN88],[JEFF85], rollback is initiated whenever a straggler arrives. Rollback means restoring the process to the appropriate state and undoing the effects of all messages since the time given by the straggler's time stamp. To enable state restoration back to that point in simulated time, each process must maintain an actions log which it uses to undo previously executed activity. Rollback includes "unsending" all messages sent (erroneously in view of the straggler message) since the straggler's time stamp, by sending an *anti-message* that annihilates the original when it reaches its destination. If a process receives an anti-message that corresponds to a message that it has already processed, then that process must also rollback. This procedure is repeated until all the effects of the erroneous computation are cancelled. Gafni showed in [GAFN85] that this rollback wave is bounded[1] under some constraints.

Gafin [GAFN88] proposed a variant of Time Warp where anti-messages are only sent for messages which are not reproduced as we perform again the computations involved in the roll back, accounting for the straggler. This method is known as Time Warp with Lazy Cancellation. This optimisation reduces the rollback overhead, because in some cases a straggler event changes only part or none of the computation of rolled back events, and consequently some of the erroneously generated messages are still correct. Notwithstand-

[1]The problem of proving that Time Warp would always make progress (i.e. for any given future point of simulated time, the simulation would eventually progress to that point) seems also a natural candidate for applying the Finite Injury Priority Argument.

ing, a weakness of this approach is that anti-messages that need to be sent (and would have been sent during state restoration in classical Time Warp) are delayed from sending until reexecution has progressed. Fujimoto [FUJI89] proposed augmenting each event with a causality record which points to all events generated by its processing. This increases the efficiency of performing rollbacks, but incurs an overhead of maintaining possibly a large number of causality records. Jefferson et al investigated a Time Warp with Jump Forward method. With Jump Forward Time Warp the state of the process is saved upon receiving a straggler message. Then the process rolls back to its state at the simulated time equal to the straggler's time stamp and processes the straggler event as before. If the state of the process is the same after processing the straggler as before, then it had no effect and there is no need to reexecute the rollbacked events. In this case the process simply Jumps Forward over them by restoring the saved state and continuing.

4.1.3 Critique of Parallel SRIP Simulation

Despite potential speedup in execution time due to parallel processing of sub-tasks on different processors, SRIP simulation suffers from several drawbacks, in addition to the obvious overhead of distributed scheduling. One of them is the extra burden on the programmer who must detect by himself/herself an opportunity for parallel execution, decompose the model into interacting subtasks executable in parallel, and deal with parallel coding and debugging. Further-more, causal relationships within a model may limit the degree of parallelism, especially in simulations applying the Chandy-Misra method [BRIN88],[BURK90], hence the number of processors that may be utilized simultaneously is restricted. The resulting under-utilization of processors can significantly reduce expected speedup.

Additional costs are connected with the synchronisation overhead, deadlock detection and resolution, and communication between subprocesses. These phenomena also lower the speedup by expending processor time on interprocess communication (IPC), and due to idle processors whose subprocesses are blocked, waiting for input from other, unfinished sub-processes. Apart from consuming processor power, IPC limits the multiprocessor architectures that can be used for SRIP simulation. For instance, in shared memory multiprocessors, intensive IPC can create a contention in the processor-memory interconnections, causing further delays and lower speedup [BOWE92]. Although fully connected shared memory architectures, such as crossbar or multi-stage networks, could be used to alleviate these problems their costs grow rapidly as the number of processors increases, and they are typically limited to medium or small scale multiprocessor systems. In fact, one can also classify concurrent simulation techniques into either *parallel simulation techniques* intended for simulations executed on multiprocessors with shared memory, and *distributed simulation techniques* intended for simulations executed on distributed memory multiprocessor systems such as the Meiko Computing Surface, or on distributed systems. SRIP simulation techniques, and in particular the S.SRIP methods, are generally not well suited for running on distributed systems (network of workstations) since substantial IPC costs.

Lastly, SRIP simulation is not fault-tolerant. If a processor (or a workstation in a network) that is running a sub-task fails, the simulation fails too, due to causality between subtasks.

4.1.4 Precision Control Methods for SRIP Simulation

Contrary to previous suggestions, e.g. in [MISR86a], methods developed for sequential statistical output analysis and precision control generally do not apply to A.SRIP parallel simulation output. In fact, common techniques including the methods of batch means and spectral analysis may produce incorrect results if applied. In an A.SRIP simulation, observations may be generated by more than one logical process, and for some techniques we discussed such as the Time Warp with Lazy Cancellation, generated observations are committed for output differing to their order of generation in simulated time. As a result, the ordering of observations outputted by an A.SRIP Simulation may differ from one executed sequentially. As an illustration, let us consider a simulation aimed at estimating cell loss probabilities at an ATM input buffer operating according to the Nested-Thresholds-Cell-Discarding mechanism [YAUP92], parallelised by decomposing the model into one logical process for handling arrivals to the buffer, and one logical process for serving cells. Observations would be of binary type, say 0 if a cell is lost, and 1 if it is successfully transmitted. In this case observations are generated both by the arrival logical process and the server logical process, yet they are not guarantied to be generated and committed in the order of simulated time of their generation.

Any distortion of a time series distorts the correlation structure of the output observation stream, so that methods for selecting a suitable batch size based on degree of autocorrelation tests [PAWL90],[MEKE84], may produce misleading results. Likewise, disordered observations would result in a distorted estimation of the spectral density function, when applying the spectral analysis precision control method, so the estimate cannot be guarantied to converge to the true spectral density function as the number of observation increases, thus the estimate of precision produced may also be biased. A solution to the problem would be to timestamp observations, and then apply existing procedures to the simulation output, after they are sorted in ascending order. However, this has the overhead of storing generation times of observations, and the processing required for sorting.

4.2 Multiple Replications in Parallel

In concurrent stochastic simulation based on the multiple replications in parallel (MRIP) approach, each processor executes an independent replication of the simulated process. The results of these replications are then averaged together. This approach can potentially be applied to any model and is suitable for execution on multiprocessors as well as multicomputer networks. As mentioned it can be shown that MRIP produces statistically better results than SRIP, in terms of the (smaller) mean squared error, if the effect of initialisation bias is sufficiently solved (or is negligible) [HEID86], and when the memory available to each processor (case of parallel execution on a multiprocesor) or each participating machine (case of execution on a network of machines) is not a limiting factor[2]. Other statistical properties of MRIP were discussed e.g. in [GLYN91], [GLYN91b], [REGO92], [HEID88], [SUND91].

[2] i.e. when the physical user memory available to each processor is sufficient to store the working set of a replication

4.2.1 Multiple Independent Replications in Parallel

The estimators based on the method of independent replications (IR) can be adapted for applications in MRIP parallel simulations. In this application of IR in parallel simulation, each processor simulates multiple independent realisations of the stochastic simulation process [HEID86], [GLYN91], [GLYN91b], [REGO92], [HEID88], [SUND91]. The results of these runs are then averaged together. We will call such a modification of IR the method of *parallelized independent replications* (PIR). Let us note that in this case, the simulation is replicated in two dimensions: we have the freedom in selecting the number of processors, and each of them executes a different (random) number of replications until a given stopping criterion is met. A simulation executing according to the PIR method is stopped when the global point estimator

$$\bar{\bar{X}}_{PIR}\,(P) = \frac{1}{P}\sum_{j=1}^{P}\left(\frac{1}{k_j}\sum_{i=1}^{k_j}\overline{Y}_{ij}\right) \tag{4.1}$$

reaches the stopping criterion for the first time. Here $\overline{Y}_{ij}$ is the mean over k_j replications executed by processor j. The basic properties of this estimator were discussed in [HEID86]. Taking into account the random size of replications, another PIR estimator of the mean, formulated after the ratio estimator of the sample mean used in regenerative simulation, was proposed in [HEID86]; see also [GLYN92]. Namely they assumed

$$\bar{\bar{X}}^{*}_{PIR}\,(P) = \frac{\sum_{j=1}^{P}\frac{1}{k_j}\sum_{i=1}^{k_j}\overline{Y}_{ij}}{\sum_{j=1}^{P}k_j}\,. \tag{4.2}$$

Theoretically, PIR estimators are statistically more efficient than IR estimators used in SRIP (i.e. when multiple processors cooperate on a single realisation of a simulated process), in the sense that PIR produces final estimates of sample means with smaller mean squared error, if the problem of the initialisation bias is properly solved. Properties of $\bar{\bar{X}}_{PIR}\,(P)$ and $\bar{\bar{X}}^{*}_{PIR}\,(P)$ have been studied empirically in [GLYN92], where the coverage of their non-sequential versions was analysed, assuming the same deterministic duration of the initial transient period and the same time of each replication, for a different number of processors. It was shown that in such applications the estimator $\bar{\bar{X}}^*_{PIR}(P)$ can yield a much better coverage than $\bar{\bar{X}}_{PIR}(P)$, and its quality improves when the degree of parallelism increases, i.e. with an increasing number of processors. In [GLYN89a] and [GLYN89b] some asymptotic properties of these estimators were studied, but their full analysis has yet to be done, especially in fully sequential versions of steady-state simulations.

The PIR estimators for MRIP simulation can potentially be applied to any model and is suitable for execution on multiprocessors as well as multicomputer networks. Fig. 4.1 illustrates the behaviour of PIR simulation and compares that with the sequential independent replications simualtion technique. In Fig. 4.1a we see that with classical independent replications, k runs (replications) are made, each run using a different random number stream. The means of the parameter of interest of each run are independent, therefore classical statistical techniques can be applied to the k means to form a point estimate, and an estimate of the precision of results. Fig. 4.1b shows that the Parallel Independent Replications case is the same as the sequential simulation, except that several processors, P_1 to P_P, are used simultaneously when making replicated runs.

PIR simulation is well suited for user transparent implementation. As an example, the EcliPse [REGO92], [REGO91], SUND91] system was specially developed for user transparent, and portable PIR simulation. Eclipse included monitors for combining an equal number of samples from each replicated instance, or for a fixed number of samples (hence a random completion time, or a random total number of samples). It was also discussed in [SUND91] that EclipSe may also be appropriate for batch means and regenerative simulation, although a batch means analysis procedure, and one for batch size determination is not provided.

There is also a possibility of using other IR-type estimators in MRIP scenario. For example, one could think about adapting estimators proposed in [HEID88] and [GLYN91] for estimating transient quantities.

Notwithstanding, the question on the choice of a proper replication size needs to be answered. The lack of a method for choosing an appropriate replication size may limit applications of PIR, since an inappropriate replication size can result in very biased estimates [PAWL90]. The main weakness of

Figure 4.1: Virtual-Real time diagram fo the method of independent replications executed on (a) a single processor, and (b) on P Processors (concurrent simulation in MRIP scenario). P_i denotes the i-th processor.

PIR methods is that they suffer from greater overhead because they lead to discarding more initial observations, as each replication must go through its own warm-up period. The number of initial data sets that has to be deleted grows linearly with the number of replications. For example in a 10 processor system where each is used to make 10 replications, 100 times the number of observations in an average initial transient period must be discarded. This situation suggested to us a novel scenario for MRIP that would be a parallel generalisation of the SA/HW method of sequential estimation : The method of Spectral Analysis in Parallel Time Streams.

4.2.2 Spectral Analysis in Parallel Time Streams

Basing on our previous experience in non-parallel stochastic simulation ([ASGA89], [PAWL88], [PAWL90a], [PAW91]) and exhaustive comparative studies of various sequential estimators conducted in Chapter 3, we propose a new technique of sequential estimation of sample means in MRIP scenario, called *Spectral Analysis in Parallel Time Streams* (SA-PTS), that is a generalisation of sequential Spectral Analysis (SA) in its HW version, i.e. the version proposed by Heidelberger and Welch [HEID82].

Following SA-PTS, P independent replications of the simulation are launched when the simulation begins; each replication is run in a time stream parallel to others. Within each replication, and for each parameter being estimated, there is an associated series of *checkpoints*, each checkpoint specifying the number of steady state observations collected. When the number of observations for the estimate of a parameter equals a checkpoint for that parameter, a local point and interval estimate of that parameter is produced, and the estimates are sent to the global control process responsible for estimating that parameter. Let the number of steady state estimates collected, by the time the i-th replication has reached the j-th checkpoint be denoted by n_{j_i}. Let us denote the local point estimate produced by the ith replication at the j-th checkpoint by $\overline(X)_i(j_i)$ and the estimate of its variance by $V_{ij} = \hat{\sigma}^2[\overline(X)_i(j_i)]$. The variance of a point estimator is estimated as the value of its spectral density function at zero frequency, obtained through a regression fit to the logarithm of the averaged periodogram of the steady state segment of the output sequence as described in Chapter 2.

Thus if P processors are used to run simulation processes, each time when a global control process is contacted, it can use up to P local estimates of the mean, $\overline(X)_{1j_1}, \overline(X)_{2j_2}, ..., \overline(X)_{Pj_P}$, and their respective variances, $V_{1j_1}, V_{2j_2}, ..., V_{Pj_P}$, submitted from different independent replications of the simulated process. The i-th local point and interval estimate (produced by the ith replication) was estimated over n_{j_i} observations. The global control process for a parameter is responsible is responsible for collecting local estimates of that parameter from the parallel replications, and computing a global estimate of the mean, and its confidence interval. When selecting an estimator for the global mean, one should bear in mind that the relative entropy of any pair of local point estimates (produced by a pair of replications) does not necessarily equal to the relative number of observation produced by the two processes. For concreteness, given that a point estimate from the 1st replicated simulation process $\overline(X)_{1j_1}$ was obtained from n_{j_1}=100 observations, and the estimate from the 2nd replicated simulation process $\overline(X)_{2j_2}$ was obtained from n_{j_2}=300 observations, one cannot assume automatically that the entropy of the second local estimate is 3 times that of the first, since each replicated process would have evolved over independent time streams.

This suggested that a global point estimate, i.e. an estimate of the mean using the P local estimates, should also take the entropy of the observations into account. Let use define

$$\hat{H}_i = \frac{\hat{R}(0)_{ij_i}}{nV_{ij_i}} \tag{4.3}$$

where $\hat{R}(0)_{kij_i}$ is the estimate of the variance of $\bar{(}X)_{kij_i}$ computed over the n_{j_i}, but *assuming that the observations were independent.*

Then the natural estimator of the global mean is $\bar{\bar{X}}_{SA-PTS*}$:

$$\bar{\bar{X}}_{SA-PTS*}= \frac{n_{j_1}\hat{H}_1\bar{(}X)_{1j_1} + n_{j_2}\hat{H}_2\bar{(}X)_{2j_2}, ..., n_{j_P}\hat{H}_P\bar{(}X)_{Pj_P}}{\sum_{m=1}^{P} n_{j_m}\hat{H}_m} \tag{4.4}$$

The estimate of the variance of this estimator would require an estimation of the second central moments of the point estimators as well. We have therefore simplified the above estimator to the following pooled estimated, indeed assuming that the entropy of each observation is the same, regardless of which replication of the simulated process that it was collected from :

$$\bar{\bar{X}}_{SA-PTS}= \frac{n_{j_1}}{n}\bar{(}X)_{1j_1} + \frac{n_{j_2}}{n}\bar{(}X)_{2j_2} + ... + \frac{n_{j_P}}{n}\bar{(}X)_{Pj_P} \tag{4.5}$$

Now $\bar{\bar{X}}_{SA-PTS}$ is a linear combination of independent random variables (since the local point estimates are obtained from independent processes), the global precision is evaluated using a weighted pooled estimator of the variance of the mean:

$$V_{SA-PTS} = (\frac{n_{j_1}}{n})^2 V_{1j_1} + (\frac{n_{j_2}}{n})^2 V_{2j_2} + ... + (\frac{n_{j_P}}{n})^2 V_{Pj_P} \tag{4.6}$$

Recall that $\bar{(}X)_{ij_i}$ and V_{ij_i} are the local estimates of the mean and its variance from the i-th replication respectively, calculated over n_{j_i} observations, and n=$n_{j_1} + n_{j_2} + ... + n_{j_P}$ are the total number of observations available at the instant when the global precision control processes was contacted. The form of Eqn. 4.5 suggests that the precision of $\bar{\bar{X}}_{SA-PTS}$ can be approximated from a Student t-distribution. Following SA/HW, each component estimator in Eqn. 4.5 has the same number, d, degrees of freedom (default value: d=7); c.f. [HEID82]. Thus, the number of degrees of freedom of the pooled estimator, obtained from P independent components, is $P \times d$. High degrees of freedom, plus variance reduction in Eqn. 4.4, leads to an expectation for narrow and reliable confidence intervals, to be tested by our coverage analysis of SA-PTS. One also expects SA-PTS to be more user-friendly, since, contrary to IR, it does not require the replication size to be determined in advance (only one replication run is executed at each processor until the stopping criterion is met), and SA-PTS saves data (fewer initial observations have to be discarded, in total, from fewer replications).

Let us note that RC-PTS, the method based on regenerative cycles and suggested in [REGO91], [SUND91] and [REGO92], although conceptually similar and simple, requires from users to select regenerative points. Proper selection of regenerative points is practically impossible without a deeper understanding of the system being simulated, and good (frequently occurring) regenerative points do not always exist. On the other hand, poorly chosen regenerative points may lead to excessively long run lengths. The quality of RC-PTS has yet to be analysed.

In SA-PTS implemented in AKAROA pooled estimates of analysed parameters and their precision are calculated by their corresponding global control processes. If several parameters are estimated during one simulation run, then one global control process is responsible for the analysis of each parameter. If the required precision of the steady state estimate of a parameter has been achieved, the global control process for that parameter initiates a termination request protocol. Subsequently, if it has determined that estimators of *all* parameters have achieved their required precision, i.e. if all global control processes have secured estimates of their respective parameters to the required level of precision, messages are sent to initiate the reporting of results, and then stop all simulation processes. Virtual-Real time interactions between the SA-PTS simulation processes in MRIP scenario are shown in Fig. 4.2 .

The design goals of AKAROA, and its fuctionalities from the user's point of view as specified by its user programming and runtime interface will be considered in the next Chapter. Then the architecture, implementation, and

Figure 4.2: Virtual-Real time interactions between processes in SA-PTS.

the results of initial performance studies of AKAROA are reported in Chapter 6.

Chapter 5

AKAROA : a Package for Automating the Generation and Process Control of Parallel Stochastic Simulation

Simulation models are essential tools for describing and evaluating new systems in engineering and computer science, and for providing us with insight on the behaviour of existing ones. Such simulations must be sufficiently detailed and be conducted with proper statistical analysis of output data to yield meaningful results, yet the execution of realistic simulation models of complex networks can take an impracticably long execution time in convention computers, or be prohibitively expensive if supercomputers are used to achieve a reasonable execution time.

With the advent of powerful and cheap microprocessors, considerable effort focused on methods for parallel discrete event simulation that might exploit the potentially enormous processing power of multiprocessors or multicomputer networks. Major approaches to parallel simulation were reviewed in the previous section. Notwithstanding, developing parallel simulators often require long lead times and expert knowledge in reasoning about parallel systems and parallel programming and debugging. This may be one of the major drawbacks of traditional parallel simulation techniques, since it may merely replace a simulation executing time bottle-neck, by a parallel simulator coding bottle-neck. Furthermore the speedup offered by those in the SRIP class of methods is dependent on the system being modelled, such that a SRIP simulator running on a multiprocessor can not perform better than on a sequential processor if the system being modelled does not have any parallelism, or if the nature of parallelism is unfeasible to exploit due to high interprocess communication costs. If simulation is used as a step in the research loop, slow simulator development and execution problems may be magnified because often several 'pilot runs' are required before a production run. In addition, once results are at hand, the knowledge gained may raise the need for new runs with a simulator of a revised system, or even the focus of the study is changed, requiring major changes to the simulator.

Given the demand for obtaining statistically valid results fast by simulation in telecommunications research, and the potentially high, system invariant speedup that can be delivered by the SA-PTS parallel simulation technique we proposed previously, combined with the need for automating parallel simulator construction , led us to develop the AKAROA simulation package.

AKAROA consists of an object-oriented simulation model construction toolkit, a set of automatic precision control objects, and a set of distributed processes for creating and managing an environment for steady state simulation based on the SA-PTS parallel estimation paradigm, as shown in Fig. 5.1. With AKAROA, a heterogeneous network of workstations and/or multiprocessors functions as one (loosely-coupled) virtual computer with the sum processing power of its member machines, from a user's perspective.

This chapter describes AKAROA as seen from the user's point of view, by describing our aims, target environments, and the user's programming and runtime interface. Specifically the purpose and aims of AKAROA are discussed in section 5.1. The main issues that we had to address when design the user interface of this package for SA-PTS MRIP simulation, and the decisions we made are explained in section 5.2. In section 5.3 the target environment of AKAROA is stated. Section 5.4 presents the user programming interface, and section 5.5 presents the runtime interface of AKAROA. As shown in Fig. 5.1, AKAROA is also contains a Build module. The Build module is an object-oriented toolkit for fast construction of discrete simulation models in

Figure 5.1: The block structure of AKAROA

C/C++. The use of Build is optional, and the user programming interface to this module is described in section ??. The architecture, implementation, testing and benchmarking, and possible applications of AKAROA will be presented in Chapter 6.

5.1 Aims and Motivations

The primary purpose of this package is to speedup simulation using multi-processors and/or multi-computers networks, thereby obtaining statistically valid results in a feasible time even for complex simulation models. We have distinguished three aims that had to be achieved to support this purpose. The foremost aim is to make parallel simulation user-friendly and efficient. Within the telecommunication research community, concurrent simulation programming is considered a difficult task, undertaken only by members of a narrow group of experts. Even researchers with substantial simulation experience find it difficult to acquire parallel coding, debugging, and testing skills needed for constructing parallel simulation models using existing software tools.

Apart from complex programming, the degree of parallelism of a distributed simulation is limited by the amount of parallelism inherent in the system being modelled [BURK90], [BRIN88]. It is therefore not surprising to hear many researchers' complaints about poor performance of their parallel distributed simulators [FUJI88], [LAZO89]. The technique of paralle independent replications (PIR) parallel simulation that was previously available (see section 4.2.1 on page 72) was easy to implement, and there were systems developed for its support, such as the EclipSe system [REGO91], [SUND91], [REGO92]. Unfortunately the fundamental question of an appropriate replication size remains unanswered, and the use of improper replication sizes can lead to strongly biased estimates. Replicating independence replications in parallel is also inefficient in the sense that each replication must traverse a transient stage, during which observations must be discarded. This can lead to a considerable wastage of simulation time.

AKAROA is based on the SA-PTS technique, whose speedup obtained from the use of multiprocessors or multiple computers is theoretically not constrained by the degree of parallelism inherently existing in the model. In SA-PTS each processor runs only one instance of the simulation model, hence this method is more efficient than parallel independent replications, in the sense that fewer observations are discarded. Also it does not suffer from replication size determination problems.

One can divide parallel simulator construction into three logical phases. To assist the user in developing and executing SA-PTS parallel simulation, AKAROA consists of three modules, that execute or fully automate the three main phases of any simulation experiment. Namely, AKAROA's Build module allows users to build simulation models in an easy and fast way, without being distracted by the burden of parallel programming, or statistical analysis and parallel precision control. Once a model is complete, the desired precision control component from AKAROA's Control module can be simply attached. Given this sequential simulator augmented with AKAROA's precision control objects, AKAROA can automatically transform the simulator into a parallel simulator. At runtime the Parallel_Simulator_Manager (PSM) of AKAROA automatically creates an environment in which parallel simulations can exist (i.e. a collection of support processes distributed among the computers). Thus, precision control, parallelizing of the simulator and parallel execution of the simulation, and distributed termination, are fully transparent to users. With AKAROA, a heterogeneous network of workstations and/or multiprocessors function as one (loosely-coupled) computer with the sum processing power of its member machines — from a user's perspective. Fig. 5.2 outlines the parallel simulator generation procedure as it used in AKAROA.

Figure 5.2: Parallel Simulator Generation with AKAROA

Designing AKAROA we had three secondary aims. Firstly, we wanted to use this package for transforming existing uniprocessor simulators into ones for parallel execution, whether they were written in C, C++, or AKAROA's Build toolkit. Secondly, we wanted still to be able to run sequential simulation, together with its various run-time precision control options. Lastly, AKAROA processes must coexists with those of other network users without hindering their progress to an appreciable extent. This AKAROA should achieve by using idle processing power whenever possible.

5.2 Design Issues Relating to the User Interface of AKAROA

5.2.1 Development Language and User Interface

Simplicity, space and code efficiency, flexibility, as well as compatibility with existing sequential simulation programs, were the main considerations when selecting a development language and designing the programming interface for AKAROA. Recognising the naturality of an object-oriented approach in constructing simulation models by means of hierarchically encapsulated classes of objects, AKAROA is written in C++. A user of AKAROA should be required to add only one extra line of code to his/her sequential simulation program before AKAROA transparently transforms it into a parallel simulator. Thus, users should not even need to be aware of the existence of multiple (parallel) simulation, precision control, and PSM processes during simulation, since their creation, location (machine and port addresses), co-operation, and inter-machine interprocess communication, are hidden from users. Binding of simulation processes to various precision control and PSM processes should be performed dynamically, yielding a flexible and fault-tolerant system featuring totally transparent parallelization from users' point of view, both in the semantic and syntactic sense.

AKAROA's sequential precision control services should be accessed by declaring an object for output data analysis, just like the objects implemented for the precision control of estimates in normal sequential simulation (see section 3.1.2 on page 50). Like normal (non-parallel) simulations, the member function responsible for precision control is later called whenever a new observation is generated. The function accepts the value of a new observation as parameter and returns one of two values (i.e. either CONT or STOP) that either orders the simulation to be continued (desired precision of estimates has not been achieved) or to be terminated (all estimates reached the required level of precision). AKAROA's implementation of SA-PTS simulation should be semantically identical to a normal non-parallel simulation; only the type of object that needs to be declared is different. The syntax for object declaration and for invoking the object's member functions should be also identical to those in the non-parallel case.

5.2.2 Level of User Involvement

Three unusual aspects of AKAROA's application interface can be summed up in one word: transparency. First, the AKAROA users need not know about the complexities of statistical analysis and precision control when writing a simulator. Major statistical decisions, such as confidence levels for estimates of various intermediate parameters, significance levels of statistical tests involved, step sizes between consecutive checkpoints of sequential estimation, etc., should be made by the designers of the package and offered as reliable (default) values for a given class of typical simulated processes, thereby allowing users to fully focus only on defining the simulated system. Once a user has written a normal simulator, the precision control services of AKAROA can be added by inserting one statement in the program. Second, AKAROA transparently transforms the resulting (sequential) simulation program, whether it is written in C, C++, or using AKAROA's BUILD toolkit, into one suited for parallel execution. Thus the user need only program the simulator for running on a classical uniprocessor. Thirdly, AKAROA offers runtime transparency, meaning that the collection of networked workstations is presented to the user as a single virtual uniprocessor, with the sum processing power of its members. A user runs AKAROA simulations in a similar manner to running any normal (sequential) program. The (parallel) simulation process can be started or stopped by the user using PSM commands, just as if they executed on a single machine. As mentioned in section 4.2.2 (on page 74), parallel simulations based on SA-PTS exists as a set of simulation processes (simulation engines) which cooperate with a set of global precision control processes (one for each parameter being estimated). Thus multiple simulation engines need to be created at runtime. When they need to contact a global precision control process that is responsible for the estimation of a specific parameter, the machine address and port number of that process must be (somehow) located, or one must be created if it does not yet exist. All such aspects of parallel execution, including distributed process creation and management, inter-machine interprocess communication, and distributed termination, are invisible to the user. The desire for runtime transparency can conflict with the advantage for providing powerful semantics. For instance, whilst most users would prefer to start parallel simulations in the same manner as sequential (uniprocessor based) runs, more sophisticated users might want to select machines to be used for a given run, and to specify 'nice' priorities for processes on each of the chosen machines, as well as certain time intervals when some of the chosen machines should not be used. We chose to resolve such friction between strong semantics and transparency following the principle: Provide the parallelism transparent form of a PSM command as its default, and offer more powerful forms of the command as options, or allow advanced users to define more powerful commands from basic ones.

Resulting from AKAROA's three module structure, see Fig. 5.2, the application interface of AKAROA can be logically divided into a programming interface (i.e. the elements a user can use in his/her program to access services of AKAROA's Build and Control modules), and its runtime interface consisting of a set of commands for running parallel simulations (calls to AKAROA's PSM).

5.2.3 Model Construction

Benefiting from the adopted object-oriented programming, AKAROA has been also equipped with an object-oriented toolkit, called Build, which allows users to work at a high level of abstraction when constructing new simulation models, either by using already existing building component types (classes) or defining new components in terms of existing ones. A component may model any entity of a given simulated system, for example a priority queue, encapsulating its attributes, and procedures for their manipulation. One immediate benefit of modular construction is that any component of the model can be readily and independently tested. Its data can be then manipulated through specific access functions, such as functions for enqueuing and dequeueing, helping to protect its consistency. Lastly, components can be easily reused, either directly in another simulation or in definitions of new component types. Of course, simulation models already constructed can be directly incorporated in AKAROA; thus, the use of Build is optional.

5.3 Target Environment

AKAROA was developed for multiprocessor systems running UNIX, or a network of heterogeneous workstations running UNIX, or a combination of both. For portability, no proprietary software or system dependent features were used. It is compatible with recent System V, Berkley 4.3BSD, and Sun OS 4.1.1 releases, or newer versions of the UNIX operating system.

5.4 User Programming Interface

AKAROA's user interface is very simple. Essentially, the programming interface consists of just one object, and the user runtime interface consist of just one command for simulation execution. The *main* elements of the user programming and runtime interface, and the stage of the simulator development and execution process in which they are used, are shown in Fig. 5.3.

Figure 5.3: Main elements of user's programming and runtime interface

5.4.1 Control Module

AKAROA's Control module provide a class for automatic statistical analysis of simulation output data and their dynamic precision control, called Spectralanalysis. Users' see Spectralanalysis as a 'black box' with one input port, and one output port. The input port accepts observations generated as the simulation progresses. The output port sends (returns) one message for each observation received on the input port. Each message indicate either that more observations are needed, or that all estimates have attained the desired precision and the simulation can stop. As discussed, an AKAROA Control object of class Spectralanalysis can be attached to an existing C, C++, or AKAROA Build program. This involves including AKAROA's header and declaring a Spectralanalysis object. Then the desired service can be requested by calling the object's member functions.

Inclusion of AKAROA's Header

All programs must have

#include ¡akaroa.h¿

added to the program beginning, before any variables or objects are declared.

Object Instantiation

The standard declaration of an object for precision control using the SA-PTS technique is

Spectralanalysis sa(max_precision, confidence) ;

where sa is an object of class Spectralanalysis. The types of *max_precision* and *confidence* is double. *max_precision* is the maximum acceptable value of the relative precision of confidence interval, see Eqn. 2.7, and *confidence* is the level of confidence. For example,

Spectralanalysis sa(0.01, 0.95) ;

creates a Spectralanalysis object for estimating a parameter to the precision of (at least) $\pm 1.0\%$ of the true value at a 95% confidence level. If Spectralanalysis is declared without parameters, then by default the accuracy of the final estimate will be $\pm 5\%$, and the level of confidence will be set to 95%. For example

Spectralanalysis sa( ) ;

creates a Spectralanalysis object for estimating a parameter with precision to within ±5% of the true value at a 95% confidence level.

Object Use

To use the *sa* object for precision control in a program, the following call should be added after the line where an observation is generated:

result = sa.processnewobs(new_observation) ;

where *result* is of type int, and *new_observation* is a double containing the value of the new observation. This function returns one of the following int values:

STOP // Stop simulation; required precision of estimate(s) estimate achieved

CONT // The desired precision has not been reached yet; continue

INTP // Process still in transient stage

ESTN // An estimation was performed, required precision not reached

The returned value is used for controlling simulation runlength. A *STOP* returned value normally indicates that a sufficient number of observations has been collected. Another possible situation when *STOP* is returned from a *sa.processnewobs(new_observation)* call occurs when the maximum simulation run length expires (without obtaining the required precision of estimates). In this case *STOP* is returned, together with a message informing about this condition sent to stdout. This is used to bound the runlength in terms of the number of observations. The maximum number of observations allowed defaults to 80 000 000, but can be changed by the user.

Typically a program would base its decision on when to stop by testing whether a *STOP* has been returned. That is, if the returned value is *STOP* then halt, otherwise continue. A common structure for a discrete event driven simulator using AKAROA's Control module is illustrated below.

```
// ————————————————————————————————————————
// Sample structure of a simulator written in C/C++ that uses the
```

```
// Control module of AKAROA for controlling the precision of estimates

#include "AKAROA.h"
// other includes and global declarations
. . . double newobs ; // value of newly generated observation
// Here is the declaration of the Spectralanalysis precision
// control object
Spectralanalysis sa( 0.05, 0.95) ;

// main program
main()
{
   int result = CONT ; // control variable initialised to continue

   ...

   // main loop : cycle through events until enough observations
   // generated
   while (result != STOP )
      {
      // process next event
      . . .
      // If new observation generated, assign its value to newobs
      // and call processnewobs like this:
      result = sa.processnewobs( newobs) ;

      ...

   } // endwhile
   // have estimate at required accuracy (of ±5%); write results and terminate.
   // Functions for querying the precision control object about results will be explained later.
   printf("Estimated cell_delay is %f. Simulation runlength is %f\n", sa.getmean(), sa.getrlength() ) ;
}
// ————————————————————————————————
```

The execution path followed by this program is depicted in Fig. 5.4. During program execution, *sa.processnewobs(new_observation)* is called whenever a new observation is generated. If the call to *sa.processnewobs(new_observation)* returns *STOP* then it means that a sufficient number of observations has been collected and the simulation can be stopped, and results written. Otherwise the simulation continues, new observations are generated and feed to the Spectralanalysis object.

The significance of the other possible returned values are the following. *CONT* denotes that more observations are needed and the simulation should be continued. As discussed in section **??**, observations gathered during the initial transient period(s) do not characterise the steady state, hence ideally all such transient observations should be discarded before starting steady state analysis. This requires and estimation of the effective length of the initial transient period. Ignoring or incorrectly estimating the duration of this period can lead to a significant bias of the final results. A return value of *INTP* indicates that the process under study is still in its transient phase, consequently the simulation should be continued. Once steady state is reached, all observations passed to the Spectralanalysis object are processed according to the requirements of the SA-PTS method, and new point and interval estimates are calculated at consecutive simulation checkpoints. Whenever the Spectralanalysis object determines such an estimate and finds that the estimator's precision is inadequate, it returns *ESTN*, otherwise the estimate is at the required level of precision and *STOP* is returned.

Querying Results

If *result==STOP* or *result==ESTN*, then the following can be used to inquire about the estimate of the parameter:

mean = sa1.getmean() ; // for the estimate of a mean

The following optional statistics are also provided:

stderr = sa1.getstderr() ; // for the standard error of the estimator

sigma = sa1.getsigma() ; // for the relative precision of the estimate

rlength = sa1.getrlength(); // for the number of observations used to
// obtain this estimate

ntrans = sa1.getntrans() ; // for the number of observations discarded
// during a given warmup period

The variables *stderr*, *sigma* and *mean* are of type *double*, and *rlength* and *ntrans* are int.

Figure 5.4: Logical execution path of simulator using AKAROA for precision control

5.4.2 Estimating Several Parameters during One Simulation Run

Usually several parameters are analysed simultaneously. If the SA-PTS method is applied, they could be analysed during a single simulation run. For instance, when simulating a packet-switched LAN consisting of ten stations, we may wish to obtain estimates of the mean delays of packets at each station, in addition to the throughputs of the stations, during the same, one simulation run only. To satisfy such requirements, a variant of the Spectralanalysis precision control object have been implemented.

Simultaneous Estimation of Multiple Parameters

Declaration

For estimating a number of parameters in a single simulation run, declare

Spectralanalysis sa(max_precision, confidence, num_parameters);

where max_precision and *confidence* are of type double and *num_parameters* is an int. *max_precison* is the maximum acceptable value of the relative precision of confidence interval [PAWL90], and *confidence* is the level of confidence. *num_parameters* is an int whose value signifies the number of parameters we want to estimate in a single simulation run. As an illustration,

Spectralanalysis sa(0.01, 0.95, 4) ;

creates a Spectralanalysis object for estimating four parameters, each to within at least ± 1.0 % of its true value with a 95% level of confidence.

Object Use

After the generation of an observation, add the following statement

result = sa1.processnewobs(new_observation, parameter_number) ;

where *result* and *parameter_number* are of type int, and *new_observation* is a double. *new_observation* should have been assigned the value of the new observation used to estimate the *parameter_number* -th parameter.

This function returns one of the following int values:

STOP	// Stop simulation; required precision of estimate(s) has been achieved
CONT	// Continue collect observations; the desired precision has not been reached yet;
ITRP	// Process still in transient stage
ESTN	// Precision at a checkpoint was estimated, but the required precision has not been reached yet; continue collecting observations
EPOK	// The estimate of the parameter parameter_number has just achieved the required level of precision. However not all estimators have the required precision. Continue simulation, but do not supply more observations for this parameter's estimation.
OKAS	// The estimator of this parameter has already achieved the required precision, and its analysis object has been freed.

Querying Results (Multiple Parameter Case)

If *result==STOP* or *result==ESTN* or *result==EPOK*, then the following member functions can be used to inquire about the estimate of a given parameter:

mean = sa1.getmean(n) ; // for the estimate of the mean of the n-th
// parameter

where *mean* is a variable of type double, and *parameter_number* is an int.

The following optional statistics are also provided:

stderr = sa1.getstderr(n) ; // for the standard error of
// the estimator of the n-th parameter

sigma = sa1.getsigma(n) ; // for the relative precision of the estimate
// of the n-th parameter

rlength = sa1.getrlength(n); // for the number of observations used
// to obtain this estimate

ntrans = sa1.getntrans(n) ; // for the number of observations discarded
// during the warmup period of the n-th parameter

The variables *stderr*, *sigma* and *mean* are of type *double*, and *rlength* and *ntrans* are int.

Dynamic Precision Specification

Another requirement, often arising when exploring a large design space by means of simulation, is to focus attention on regions of the design space where a critical parameter is within a zone of interest. For example, if we are interested in the behaviour of a LAN under congestion, we may be interested in studying the case when the mean packet delays of stations exceed a given threshold. As a result, we could save the simulation time, if our simulation experiment can be stopped as soon as we are confident (in a statistical sense) that mean packet delays are outside our zone of interest. To satisfy such requirements, several new forms of the Spectralanalysis precision control object have been introduced.

Declaration

For dynamic precision specification, when the required precision varies with the estimated value of the parameter, we should declare a Spectralanalysis object like this:

Spectralanalysis sa2(max_precision, confidence, constraint, threshold, num_parameter);

where *max_precison* and *confidence* are defined as above. In this example we chose the name sa2 for the instance of the object of type Spectralanalysis. *constraint* is either *GEQ* or *LEQ* denoting greater or less than or equal to, respectively. If *constraint==LEQ,* the meaning is that if the estimated parameter is less than or equal to *threshold*, precision control is abandoned, otherwise the required precision is smaller or equal to *max_precision*.

Object Use

To use the sa2 object for precision control in our program, add the following call to the program after the generation of an observation

result = sa2.processnewobs(new_observation , parameter_num) ;

where *result* is of type int, *new_observation* is a double, *parameter_num* is a double containing the value of the new observation, and *parameter_num* is an int containing the index of the parameter being estimated. This function returns one of the following int values:

STOP // Stop simulation; required precision of estimate(s) has been achieved

CONT // Continue collect observations; the desired precision has not been reached yet;

ITRP // Process still in transient stage

ESTN // Precision at a checkpoint was estimated, but the required precision has been not reached yet; continue collect observations

SABP // An estimation was performed, abandon precision control (stop).

If *result==STOP* or *result==ESTN* or *result==SABP*, then the functions introduced for objects related with single parameter estimation, can be also used to query results.

5.4.3 Random Number Generator Selection

During runtime, each activation of an AKAROA simulation program (simulation engine) must be driven by a unique random number stream. Users have the option of choosing the size (period) of these generators, and consequently the maximum number of simulation engines that can be employed on a simulation run.

When launched, each simulation engine first contacts and registers with AKAROA's Directory_Central process. One function of Directory_Central is to allocate unique (virtual) random number generators by dispensing seeds to each simulation engine. These mutually uncorrelated virtual random number generators were produced by extracting seeds from AKAROA's base multiplicative congruential generator which have a period of $2^{31}-1$, each seed pointing to a location in a random number sequence that is spaced from that of other seeds. For this reason, the smaller the size of virtual generator selected, the greater number of simulations can be used. The default generators used have a MEDIUM period of approximately 100000000, enabling up to twenty two simulation engines. Alternative generator choices, and their respective sizes are listed in Table 5.1. Suppose that we have declared sa1 to be an object of type Spectralanalysis. Then if we wish to use generators other then the default ones, we should invoke

sa1.set_generator(type) ;

after declaring the sa1 object, before any calls to the random number generator. *type* can be any of those listed in Table 5.1. As we can see from Table 5.1, currently up to forty four simulation engines can operate in parallel when each is using a generator which have a period equal to approximately 48000000. These "virtual" generators are defined by seeds obtained by segmenting the random number stream generated by AKAROA's base multiplicative congruential generator which have a period of $2^{31}-1$, into 44 disjoint sub-sequences.

One option for future research is the evaluation of additional random number generators suited for AKAROA's use. Incorporating more mutually uncorrelated generators in AKAROA would enabling the use of more simulation engines in parallel. By mutually uncorrelated, we mean that if we concatenated the sequences of pseudo-random numbers from each generator into a "macro-sequence", then the macro-sequence should also quality as a random sequence. The quality of the generators with respect to SA-PTS would include the degree in which they are mutually uncorrelated, in addition to randomness of individual generators, and their computational efficiency and period.

Checkpoint Distribution

Within each simulation engine, and for each parameter, spectral analysis is performed to obtain an estimate of the precision of the local estimate at cheakpoints. Checkpoint locations are indicated by the number of steady state observations generated for the parameter by the simulation engine. By default, the distance between checkpoints increases geometrically where the first checkpoint is determined from the nature of the simulated process, thus following the well tested procedure in [PAWL91]. Alternatively, constant distances may be desired. To adopt this option, include the statement

sa1.set_checkpoint_type(CONST) ;

Table 5.1: Selection of random number generators in AKAROA

following the declaration of the Spectralanalysis object. Both checkpoint distribution options were evaluated, and their impact on the quality of estimators, and on the execution overhead, and the amount of communication required between machines are reported in Chapter 6.

5.4.4 Programming Interface for Parallel Simulations

To transform an existing sequential simulator which uses the spectral analysis object of AKAROA Control for precision control of estimates, into a simulator suited for parallel execution using the SA-PTS methodology, only a single word in the program needs to be capitalised. This is done simply by changing declaration of the precision control from

Spectralanalysis sa1(...) ;

into

SPECTRALANALYSIS sa1(...) ;

No other changes are necessary.

There are many attractive aspects of this 'invisible' parallel simulation manager interface. One is that the resulting parallel simulation program would appear essentially identical to the ordinary sequential simulator, both syntactically and semantically, no matter which of the various precision control options are used, or whether we are estimating one or multiple parameters. This totally shields the user from the complexities of parallel programming and debugging, giving the user an illusion of a normal sequential program, thereby achieving our first aim for AKAROA. The advantage is that the user can concentrate on building a correct sequential simulator, the transformation to a parallel simulator being overhead free. For instance, recall from section 4.2.2 that a parallel simulation based on SA-PTS exists as a set of simulation processes (simulation engines) which cooperate with a set of global precision control processes, one precision control process for each parameter being estimated. Yet, the user need not be concerned with nor aware of the need for, or existance of the simulation and precision control processes, when writing a simulation program. During runtime, is a simulation engine need to contact a global precision control process, the machine address and port number of that process must be (somehow) located. Notice that the name of the machines where the global precision control processes might exist, and their port numbers, do not have to be specified. If the estimation process which is responsible for an estimator doesn't exist, one must be created, and this is also handled transparently. Also at runtime each simulation process must be driven by a unique random number stream and this too is done automatically. Moreover, possible faults in the communications network between a processor running the simulation instance and another one running the estimation process must be detected and resolved. These tasks are handled invisibly, so the user is not attend to them during the programming phase.

Another significant advantage is that any existing simulator whether written in C, C++, or a package that enables the import of C/C++ routines can easily be augmented with AKAROA's precision control objects for runlength control and hence be automatically transformed into a simulator suited for parallel execution.

Generating Reports

AKAROA automatically reports simulation results. Thus querying results, formatting and writing to file(s) is performed for the user, if the parallel simulation option is selected, and therefore need not be programmed by the user. Two files are produced. One is intended for reading and contains a 'pretty' report of each parameter being estimated. Fig. 5.5 shows a sample report. The other file contains results in a space delimited format for direct import into a graphing package, thereby speeding the graphing of results. Currently graphing packages or packages with graphing facilities that accepts files in that format include Kelidagraph, Deltagraph, Cricket Graph, Statsview, and Microsoft Excel. All graphs reporting the benchmarking of AKAROA (Chapter 6) of this report were produced using data imported from AKAROA generated files.

The default names of the 'pretty' and export format report files produced by AKAROA are "akreport.dat" and "akreport.sum" respectively. User specified names can be used too.

Figure 5.5: Report generated by PSM for a parallel simulation of an $M/M/1/\infty$ system using 6 processors

Filename Specification

If the user prefers to give special names to the report files generated by AKAROA, the following member function is provided,

sa1.set_result_fnames(fname1, fname2) ;

where *fname1* and *fname2* are of type *char ** pointing to the names (strings declared as char [40]) preferred for the report and export data files respectively.

5.5 User Runtime Interface

With AKAROA, running a parallel simulation on a network of workstations is essentially identical to running a (sequential) simulation on one machine. Also AKAROA's PSM provides commands for managing the distributed SA-PTS processes, which have similar semantics to those provided by UNIX which operate on one process on the local machine.

5.5.1 Default Simulation Startup

AKAROA PSM's functions for initiating a parallel simulation are invoked using PSM's *akkey* command, followed by the name of the simulation generated by AKAROA, and its parameters. That is

akkey "<simulation_engine_name> <optional parameters>"

initiates distributed execution of a parallel simulator generated by AKAROA, that ordinarily (i.e. without selecting the parallel simulator option of AKAROA) would have being started using

<simulator_name> <optional parameters>.

As an illustration, suppose that we have a ordinary simulation program using the Spectralanalysis object for precision control, whose run-file is called 'dqdb', and expects the number of dqdb stations, and the normalised load as parameters. Prior to being transformed into a simulator for parallel execution, the (sequential) simulator would be executed like

dqdb 10 0.80

for simulating a 10 station dqdb network under 80% normalized offered load. After transforming the simulator into one for parallel execution (i.e. by replacing Spectralanalysis with SPECTRALANALYSIS in the object declaration in the program), it should be executed by invoking PSM's akkey as follows

akkey "dqdb 10 0.80"

Notice that the semantics of parallel simulations (started using akkey) are identical to that of the original (sequential) simulator. Thus the akkey call is completed only when the parallel simulation has finished. Like the sequential counterpart, the user can use '&' to place the execution of the parallel simulation command in the background. For example

akkey "dqdb 10 0.80" &

is valid.

5.5.2 Specifying the Degree of Parallelism

The user may specify the maximum number of machines that could be used for running a simulation using

akkey "normal_command_string" num_machine

option, where "normal command string" denotes the <simulator_name> followed by any <optional parameters>, as usual. Here num_machine is an integer value of the maximum number of workstations that should be used. PSM will then use max(num_machine, num_alive_members) workstations for the execution of AKAROA processes, where num_alive_members is the number of machines in the AKAROA domain that are in state AVAILABLE. A machine in the AKAROA domain is said to be AVAILABLE if it is ready to execute AKAROA processes. Machines may not be in AVAILABLE state because they cannot be contacted by PSM's directory_central process, due to a network problem, or machine failure. When *akkey* is used with the num_machine option, it will report the number of machines chosen for executing simulation processes, then blocks until simulation completion, then terminates.

Selecting Participating Machines

One can also specify the machines that hosts the distributed SA-PTS simulation processes.

*akkey "normal_command_string" [machine_name]**

executes the parallel simulator specified by "normal_command_string" using all machines in the [machine_name]* list that are in the AVAILABLE state. To illustrate,

akkey "dqdb 10 0.80" whio mohua titi kiwi kaka &

executes the parallel dqdb simulator using machines with hostname whio mohua titi kiwi kaka.

Specifying Process Priority

Sometimes it may be useful to specify the 'nice' priority of AKAROA processes that run on given machines. The 'nice' value of a user process is one of the parameters considered by UNIX's kernel for deciding a process's execution priority. Nice values can be used to achieve a degree of load balancing. The option for specifying nice values for processes on machines that would be used in a given simulation run is

*akkey "normal command string" [machine_name nice_value]**

where nice_value is an integer between 0 and 19.

Running a Series of Parallel Simulations

One can specify a series of simulations that are executed in sequence using a shell script, in a manner to that for running a sequence of simulations on a single machine. For example, a script containing

akkey "dqdb 10 0.80" whio mohua titi kiwi kaka
akkey "dqdb 20 0.80" whio mohua titi kiwi kaka
akkey "dqdb 30 0.80" whio mohua titi kiwi kaka moko
akkey "dqdb 40 0.80" whio 8 mohua 8 titi 19 kiwi 19 kaka 19 moko 19

would run four parallel simulations in sequence. The last two runs may use one more machine (moko), and processes of the last run would have higher 'nice' priorities as specified.

5.5.3 Domain Creation and Maintenance

PSM provides commands for the user to query the status of the AKAROA domain, and for adding and removing of machines from the domain, and for changing the membership class of machines. The command

akstat

produces a report on the machines in the AKAROA domain, their membership class, and their status.

5.5.4 Registering New Machines

A machine can be added to the AKAROA domain using

alaunch <machine_class> &

where the <machine_class> parameter specifies the machine as a class 1, 2, or 3 machine. The class of a machine determines the default nice priority of (most) AKAROA processes hosted by that machine. Additionally, a machine with a higher membership class would be chosen for participating in a simulation run before machines of lower class. This ability to distinguish three classes of machines was motivated by our need to segment machines in our AKAROA domain based on their ownership. At the University of Canterbury, we have access to VAXes and a SUN server which belongs to our computer services center. These machines are shared by many departments, and therefore should be used least, and AKAROA processes on them that are CPU intensive should be executed at a high nice level. Hence they were registered as class 3 machines. We also have access to teaching machines within our department. These we can use more frequently, though our processes should still have their priorities lowered. This category of machines were therefore registered as class 2. Lastly, we have access to a machine dedicated for our simulation research, which naturally fits as a class 1 machine. Distinguishing machines into classes helps AKAROA to match machine usage with their political status, by default.

Changing Membership Class

Invoking

alaunch ¡new_membership_class¿

on a machine already registered with AKAROA will change its current membership class to new_membership_class. Also an updated report of all machines in the AKAROA domain will be displayed.

5.6 Simulator Construction Toolkit: The Build Module

AKAROA's Build Library is an object-oriented toolkit for the construction of discrete-event simulation models in C/C++. Discrete-event simulations view system models as structured collections of *entities* bounded into webs of *relations* and transformations [KREU86]. Activities cause changes to the system's state at discrete points in time. When the system changes state we say that an *event* has occurred. *Queuing* often occurs when more than one entity contend for the same facility (entity). In fact, although a wide variety of systems can be studied using discrete-event simulation techniques, the queuing network paradigm is the most popular one [KREU86]. Lastly many processes that we want to simulate involve some randomness.

Accordingly, to support discrete event driven simulator development AKAROA Build provides three basic classes of objects: entities, queues, and an event scheduler, as well as supporting functions for representing stochastic behaviour. Concepts behind AKAROA's Build were based on a Pascal simulation toolkit reported in [Vignaux]. In this section we introduce each of these elements of discrete event simulation, and the corresponding object provided by the AKAROA Build Library for its modelling.

5.6.1 Entities, Interrelationships, and Events

Looking at a real world system, we see that there are certain distinct objects, each of which possesses properties of interest. Additionally there are certain interactions occurring in the system that cause changes in the system. We use the term *entity* to denote an object of interest in a system, and the term *attribute* to denote a property of an entity. Of course an entity may have more than one attribute. Discrete event-driven simulation views activities as causing changes to the system's state at discrete points in time. When the system changes state we say that an *event* has occurred.

5.6.2 Example 1: Bus Transportation System

In describing a public bus transportation system, the entities of the system are the (fixed number) busses, and the passengers. The attributes of busses are such factors as the maximum number of passengers the vehicle could carry, and its speed. A passenger's pertinent attributes would be whether he or she is an adult or child, his time of arrival to the bus stop, and his destination. Events which may occur in this system might be the arrival of a bus to a bus stop, the arrival of a passenger to a bus stop, and the departure of a passenger from a bus. There are interrelationships between the entities which may be of relevance to the study. Each bus is associated with a group of passengers through the passengers being onboard the bus. Let us say that two passengers are 'co-travellers' if both are onboard the same bus. Give knowledge of 'onboard' relationships between busses and passengers, we can also deduce the (transitive) 'co-travellers' relationships between passengers.

5.6.3 Example 2: ATM Congestion Control

Consider an ATM switch with output queuing. Cells arriving are switched to the appropriate output ports where they join cell queues with finite capacity, to wait for transmission on outgoing links. The purpose of the simulation might to determine the loss probabilities of each class of cells. Here the entities would be the ATM cells, their relevant attributes to the study would be their arrival times to the switch, their destination (which determines the output queue they will join) and their priority class. Relevant events would be cell arrivals to the switch, and cell departures from an output queue.

5.6.4 Entities

Entities model real world objects and their interrelationships. They can also join queues, and be associated with a scheduled event. For example, customer entities may join queues. Naturally a customer may be in more than one queue at ant time. If a DEPARTURE event was scheduled, then it is handy to associated the customer (entity) that corresponds to the event with its occurrence, namely the customer that will be departing when that event occurs. We now describe the declaration of entities, setting and querying of their attributes, mechanisms for associating one entity with (a group of) others, and for associating them with scheduled events.

Declaration

Create entities in your programs by declaring them statically (semi-statically) or dynamically. To illustrate:

ent my_entity = ent(my_id, my_num_attribute, my_quant_attribute) ;

will declare my_entity to be an entity and initialise its identity_attribute to my_id, its num_attribute to my_num, and its quantity_attribute to my_quant_attribute. my_id and my_num_attribute should be of type int, and my_quant_attribute should be a double. Thus the function ent(int, int, double) constructs an entity and assigns its attributes. Create an entity dynamically by

*ent * ent_ptr ;*
ent_ptr = new ent(my_id, my_num_attribute, my_quant_attribute);

For concreteness, consider the ATM simulation. Here entities may model ATM cells arriving to a switch. We want to estimate the cell loss probability and queuing delay. Hence relevant attributes of an incoming cell would be its priority class (used in discarding decisions), its destination (used to determine which output port it will be routed to), and its arrival time (used to calculate its delay in the switch). A declaration of an entity to represent an ATM cell might proceed like this:

*ent * cell_ptr ;*
// determine priority, destination, and time of arrival of next cell
// assign them to (int) priority, (int) destination,
// and (double) arr_time resp.
cell_ptr = new ent(priority, destination, arr_time);

where cell_ptr is an entity pointer holding the address of the entity representing the new cell. The cell entity will have its identity, num, and quant attributes initialised to priority, destination, and (double) arr_time respectively.

Attributes

After creation, an entity's attributes may be reassigned, or queried using the following member functions:

void ent_ptr-¿assignident(int id) ;

sets the identification of the entity pointed to by ent_ptr.

int identity = ent_ptr-¿ident() ;

gets the identity of this entity.

void assignnum(int num) ;

assigns an integer attribute to this entity.

int its_number = ent_ptr-¿number() ;

returns the integer attribute of this entity.

void assignquant(double qty) ;

assigns a double attribute to this entity.

double its_quant = ent_ptr-¿quant() ;

asks for the value of the double attribute of the entity.

5.6.5 Entity Interrelationships

Representation

An entity may be *associated* with other entities. For example given

```
ent * bus1 = new ent(1, 40, 30.5) ; // bus1 is a bus entity
ent * Jack = new ent(20, 1, 1600) ; // Jack is a person entity
ent * Jill = new ent(20, 1, 1600) ; // Jill is a person entity
```

we can represent the 'onboard' association, i.e. that Joe is onboard bus1 using

bus1-¿tie(Jack) ;

And if Jill now boards the bus1 this can be represented by

bus1-¿tie(Jill) ;

Querying

Having represented relationships, we may later query on possible associations, for example to answer "Who was the last to board bus1 ?" we should use ent * last_passenger = bus1-¿tailtied() ;

Entities associated with a given entity using tie are stored internally in a queue within the entity object. The bus1 object will therefore contain a queue of entities. This queue now has two entries, a pointer to Jack and a pointer to Jill. Build's entities allows direct access to its queues for querying through

gqueue * bus1_tied_queue = bus1-¿tied() ;

With bus1_tied_queue questions such as

Is Jack onboard bus1 ?
How many passengers are onboard bus1 ?
Is Jack onboard the same bus as Jill ?

an be easily answered using the queue search functions in the next section.

Changing Associations

In addition to queue operators, the following can be used to change associations:

void bus1-¿untie() ;

releases the last entity tied to entity

ent * bus1-¿tailtied() ;

gives pointer to the last entity associated with bus1

void bus1-¿entreport() ;

// prints attributes of the bus1 entity, and recursively prints the attributes
// of entities tied to bus1 and that for entities tied to them.

5.6.6 Queuing

Queuing often occurs when more than one entity contend for the same facility (entity). In example 1, passenger entities queue for use of bus entities at bus stops. The rule by which passenger entities go from the queue to the bus entity might be FIFO. In example 2, incoming cells to a switch queue for transmission on the outgoing link, but following a more complex discipline to satisfy the different quality of service requirements of each class of cells. In fact, although a wide variety of systems can be studied using discrete-event simulation techniques, the queuing network paradigm is the most popular one [KREU86].

Basic queue classes in AKAROA Build are the FIFO and Generic queues. Derived queues include NTCD, NTCD/SE and DQ-NTCD queues used by the ATM simulators.

We can declare a FIFO queue as follows:

fifoqueue queue_name(queue_type) ;

where queue_type is an int and equal to 1, or 2, or 3. If queue_type is one, then the queue will be a queue of entities. If queue_type is 2 then the queue will be an integer queue. If queue_type is one, then the queue will be a queue of doubles.

As expected, there are four member functions defined for a FIFO queue of entities:

*addtail(ent * e)*

adds an entity (pointer) to the tail of the FIFO queue. E.g.,

queue_name.addtail(ent_ptr) ;

enqueues the entity pointed to by ent_ptr to queue_name. When enqueueing, only a pointer to the entity, not the entity itself is added to the queue. In this way an entity may be a member of many queues.

headof() returns a pointer to the entity at the head of the queue. E.g. for an entity FIFO queue,

*ent * queue_head = queue_name.headof() ;*

returns the a pointer to the entity at the front of queue_name, or NULL is returned if queue_name is empty.

tailof() returns a pointer to the item at the tail of the queue. E.g. for an entity FIFO queue,

*ent * queue_tail = queue_name.tailof() ;*

returns a pointer to the entity at the end of queue_name, or NULL is returned if queue_name is empty.

lengthof() returns an int denoting the length of the queue. E.g.,

int qlength = queue_name.lengthof() ;

returns the number of entities en queued in queue_name to qlength.

behead() removes (de queues) the entity at the head from the queue. For example,

queue_name.behead() ;

deletes the entity at the head of queue_name. Its length will therefore be one less.

Generic Queues

A generic queue can be declared like:

gqueue gq_name(int gtype) ;

where gtype is either 1,2, or 3 as before. For entity gqueues, the following operations are available, in addition to those that can be used on FIFO queues:

```
void betail() ; // deletes the entity from the tail of the queue.
void addhead(ent *new_ent) ; // adds new_ent to the front of the
                             // queue.
void qdelete(ent *e) ; // Searches for an entity pointed to by ent
                 // in the queue. Delete (dequeue) it if found
ent * nthof(int n) ; // Returns pointer to the n-th
                       // entity enqueued.
item *qsearch(int eid, ent *etied) ; // find item with entity
                 // having id eid and to which ent etied is tied
item *ordqsearch(int eid, int order); // find a queue item whose
            // entity have an identity number satisfying the desired
            // order constraint. order can be GTN, GEQ, LTN or LEQ
```

5.6.7 Scheduling Events

A scheduler called 'st' is provided by AKAROA. It can be used by calling :

*void st.schedevent(int eventnum, double delay, ent *e) ;*

This function is used to schedule an event. *eventnum* is the event number, that is a number you chose to uniquely identify a type of event. *delay* is the length of time before eventnum occurs. It is the time between when it was scheduled and when it should occur. *e* is a pointer to the entity tied to the event. For example

st.schedevent(DEPARTURE, 50.20, eptr) ;

schedules a DEPARTURE event to occur 50.20 time units from the current simulated time, and associates eptr with this event.

*void st.nextevent(nevnt *retptr) ;*

returns the next scheduled event to occur, and the entity tied to that event through a nevent struct. nevent was defined like this

```
struct nevent
{
    int etype ;
    ent * etied ;
}
```

For the current simulated time, use

double time_now = st.currenttime() ;

which returns the current simulated time. In some systems, certain events may cause events scheduled in the future *not* to occur. For example, when a customer begins service, we may schedule its departure. However if pre-emption is allowed, for example if a newly arrived high priority class customer is allowed to preempt customers of a lower class already in service, then we would have to cancel the departure scheduled for the preempted customer. To cancel an already scheduled event, use

void cancel(int event, ent etied) ;*

which cancels the closest scheduled event with entity etied associated.

5.6.8 Stochastic Processes

Many processes that we attempt to simulate involve some randomness. In example 1, passengers arrive at bus stops at random times. The time taken for a bus to travel between two bus stops is also random. In example 2, cell interarrival times are random, and we may wish to assume that their choice of output ports (as determined by their destination) is determined probabilisticly.

Statistical support functions available in AKAROA Build are listed in Fig 5.6, together with a listing of other Build facilities.

Figure 5.6: AKAROA Build's Nucleus

5.6.9 Example NTCD/SE ATM Congestion Control Mechanism

Users of Build work at a higher level of abstraction, using its base components, or defining new components in terms of other components, when building the simulation model. For instance, programming the NTCD/SE simulator [YAUP92] [YAUP92a] simply involved defining the NTCD/SE queue in terms of Build's queue components, and actions to be performed when new cells arrive and/or when their service finishes. As a result, the NTCD/SE model required only a few pages of code (given below), yet the final simulator still enjoys the power and efficiency of C/C++ programs.

Chapter 6

Architecture, Implementation, and Performance Evaluation of AKAROA

In this chapter the architecture, implementation, and testing and benchmarking of AKAROA are presented. Specifically the Implementation Issues and design choices in the SA-PTS context are discussed in section 6.1. Section 6.2 presents AKAROA's architecture and implementation. The performance of parallel simulators generated by AKAROA has been tested and bench-marked by a series of 1600 experiments. Results are reported in section 6.3.

6.1 Implementation Issues in the SA-PTS Scenario

6.1.1 Distributed Process Creation and Management

As mentioned in section 4.2.2, parallel simulations based on SA-PTS exists as a set of simulation processes which are mainly responsible for generating observations (simulation engines) which cooperate with a set of global precision control processes (one for each parameter being estimated) which are mainly responsible for collecting local estimates and using them to form a global point and interval estimate. Thus multiple simulation engines need to be created at runtime. When they need to contact a global precision control process that is responsible for the estimation of a specific parameter, the machine address and port number of that process must be (somehow) located, or one must be created if it does not yet exist. The general responsibility of the parallel simulation manager (PSM) of AKAROA is to create and maintain an environment in which SA-PTS processes can exists. In addition, PSM must achieve its tasks in a way that maintains runtime parallelism transparency : the AKAROA system should appears as a single uniprocessor based machine to its users. PSM is responsible for creating non-existing, and locating existing control and simulation engine processes (SEs), allocation of pseudo-random number generators to SEs, selecting their host machines, providing facilities for communication between co-operating SA-PTS processes, and for ascertaining whether stopping conditions are met, and for distributed termination.

In designing PSM, we address the following:

- How does a given SE process specify the types and instances of the Global_Control processes it needs to cooperate with ?
- Given that a SE can identify the Global_Control processes it want, how then could that SE ascertain where (i.e. which machine address, port number) those Global_Control (GC) processes could be contacted ?
- What if the desired GC process does not currently exist ?
- How could the simulation processes know when all conditions for termination are satisfied, and how can the subset of SEs and Global_Control processes to be terminated be identified and found ?
- To implement a simulation pause, resume, or termination operation, we must have some way of naming the group of processes to be suspended, resumed, or stopped.
- How can the SE, GC and other supporting processes be created when needed ?

If all AKAROA processes executed on a single machine, then several UNIX mechanisms can be used to solve these problems of process naming, location, and creation. For example a parent process could create lines of communication (e.g. a pipe) with its child through I/O table inheritance by its child. Alternatively, the single process name space within a machine means that one process can uniquely identify another by its process id, and cooperate with that process by invoking various kernel calls (e.g. through the exchange of signals). Unfortunately, neither of these solutions are applicable to AKAROA, due to the absence of a network wide process name space, and that none of the versions of UNIX envisioned for AKAROA support any form of remote fork, nor process migration.

Early on in AKAROA's design, we dismissed that possibility of including in the SE, GC, and the other supporting programs, the network machine addresses of the workstations with which they need to communicate. This compile time binding of cooperating processes is undesirable, as any addition or removal of machines from the AKAROA domain cannot be accommodated without changing and recompiling several program code. Moreover, this would not allow AKAROA to allocate processes to machines based on their current load at runtime. We also felt that it would be conceptually neater, especially for testing purposes, to isolate knowledge about the machines and network on which the SA-PTS processes might execute, from their program code—such information is exogenous to the simulator, just as the machine identity on which a normal (sequential simulator) might execute need not be known to it.

Efficient distributed process management is another requirement for SA-PTS implementation. We discarded the possibility of using the rex, rlogin, or rsh facilities for remote process creation. One reason for this decision is that these calls are not supported by many UNIX variants, thus their use would have lowered AKAROA's portability. The main reason why we rejected their use is that they are very inefficient for the operations we wish to perform. To illustrate, to suspend a process on a remote machine using rsh, the invoking process must wait after making the call for a local rsh process to be created, and then wait while a request is transmitted to the remote machine for a remote rsh peer to be created, and then wait while environmental information is exchanged over the network, and until the remote rsh process invoked the kill call, and until the result has been returned to the local rsh process. We feel that the overhead associated with 'rsh' is too high, because many features provided by rsh are unnecessary for operations that are frequently used by AKAROA.

6.1.2 Inter-Machine Inter-Process Communication

Development of an efficient, portable and flexible Interprocess Communication (IPC) subsystem of PSM was regarded as the critical factor for achieving high efficiency of AKAROA. It is known that a careless implementation of IPC can result even in a negative speedup of parallel processing, if high IPC overhead is generated [U.MARY Tech Report]. UNIX facilities for implementing IPC mechanisms include streams, pipes, socket-pairs, and various types of sockets [QUAR85].

AKAROA's IPC subsystem must support communicating processes located on different machines, and possibly belonging to different file systems. Having considered basic IPC facilities, we chose an inter-machine interprocess communication mechanism that is based on UNIX Internet domain datagram-type sockets that allow fully file-less exchange operations. The IPC model selected for AKAROA is structured similar to a Remote Procedure Call (RPC) mechanism ([BIRR84], [BRIA90]). Selection of RPC was motivated by the fact that it has simpler semantics than the Rendezvous model [GEHA88], and that, as a higher level construct, it better encapsulates (simulation engine, control process) interactions than alternative solutions based on point-to-point message-passing [HOAR78]. The RPC modelled IPC subsystem resulted in a simpler user interface and improved efficiency of AKAROA.

6.1.3 Termination Protocol

Each Global_Control processes is responsible for computing the estimate of one parameter, but the simulation should be stopped only when all estimates are at a required level of precision. The question then is which simulation process(es) should have the responsibility of deciding when the simulation could be stopped. In answering this question, we considered solutions based on some form of status-board, voting, joint agreement, and a common manager.

6.2 Implementation

6.2.1 Basic System Structure

As mentioned, AKAROA consists of three modules to automate parallel simulation construction and execution: Build, Control and Parallel-Simulation-Manager. The role of each module in the construction and execution of parallel simulations with AKAROA is illustrated in Fig. 6.1. The Build module enables users to focus on defining the logic of the system to be simulated, thus freeing him/her from the burdens of event scheduling, run-time control and statistical output data analysis. With Build, users work at a high level of abstraction, using building components (classes) or defining new components in terms of other components, when building the simulation model. Once completed, the desired precision control method supported by Control module can be simply attached. For parallel execution, the SPECTRALANALYSIS precision control object would have been chosen. In designing AKAROA we decided to engineer the parallel simulation manager components to support user access using existing C/C++ syntax. A beneficial consequence is that full transparency in generating parallel simulators is achieved, that is if parallel simulation is desired, a user simply uses the SPECTRALANALYSIS object instead of a Spectralanalysis object. In a user's eye, the two objects have the same member functions for use that are called in the same manner syntactically, and their functions identical. Obviously the user would not need to learn new non-C++ syntax to use AKAROA, but also no pre-compiling is required, allowing for faster program development.

6.2.2 Control: PSM Gateway

As shown in Fig. 6.1, having attached the SPECTRALANALYSIS precision control object, the sequential simulator given by the user is used to generate the code of a parallel simulation engine. At runtime, multiple instantiations of the same simulation engine code is executed, typically one simulation engine per participating machine. Each simulation engine cooperates with AKAROA's PSM processes in performing the simulation. These PSM processes include global control processes, local manager processes, and a directory_central process and the akkey process.

Simulations Engines

A simulation engine is an activation of an AKAROA simulation engine program constructed with the Build, Control and PSM components. Each simulation engine is .composed of four logical units :

1. the basic sequential simulator (specified by the user),
2. a local precision analysis unit,
3. a parallel simulation manager interface unit, and
4. a management unit.

Figure 6.1: Construction and execution of parallel simulations with AKAROA

From Fig. 6.2 we see that the precision analysis, parallel simulation manager interface, and management units are encapsulated in the SPECTRALANALYSIS precision control object.

The precision analysis unit consists of a set of Spectralanalysis objects that are created at run time, with their memory allocated when needed, and freed when their function is complete. In turn each Spectralanalysis object is composed of

Figure 6.2: Structure of a Simulation Engine

a detect_initial_transient (sub) object which is used to estimate the length of the transient period of the process corresponding to the parameter being estimated for that instance of the simulated system, and a spectralanalysis (sub) object responsible for local precision estimation using an extension of the spectral analysis methodology. Similarly, the parallel simulation manager interface unit consists of a set of *sapts* objects that are created at run time, with their memory allocated when needed, and freed when their function is complete. sapts objects are gate-ways to AKAROA's PSM. They enable a simulation engine process to interact directly or transitively with other processes of PSM that are distributed among workstations registered with AKAROA, in support of parallel simulation based on SA-PTS. Hence each sapts object instance of the parallel simulation manager interface unit provides facilities for calling, argument packing and fetching, and transmission and receiving from PSM processes such as the Directory_Central process and the Local_Managers of PSM, and the Global_Control processes. The other logical unit encapsulated by the SPECTRALANALYSIS precision control object is the management unit. The management unit comprises of member functions of the SPECTRALANALYSIS object. These are methods dedicated for manipulating the data and to 'drive' the (sub) objects of the SPECTRALANALYSIS object to perform the task of offering a precision control service to SPECTRALANALYSIS's caller that is semantically and syntactically identical to that provided by its sequential simulation counterpart, thereby shielding the user of SPECTRALANALYSIS from activities it conducts during SA-PTS parallel simulation execution. Fig. 6.3

summaries the logical structure of the SPECTRALANALYSIS object.

Figure 6.3: Logical Structure of the SPECTRALANALYSIS Object

6.2.3 PSM Run Time

At runtime multiple simulation engines run in parallel, along with other AKAROA processes which manage, create (launch) global control processes and handle interprocess communication. AKAROA's Parallel-Simulation Manager (PSM) automatically creates and maintains an environment which supports SA-PTS (sequential estimation by means of spectral analysis in parallel time streams, see Chapter 2 and 4). The PSM hides from the user, the fact that the simulation executes as multiple processes on different machines.

Simulation execution can be divided logically into :

1. a launch phase,

2. a simulation engine/ control process binding phase,
3. data generation and analysis phase,
4. and a termination phase.

The purpose of each phase, and role played by the PSM processes for achieving the stated purposes, are described next.

Simulation Engine – Control Process Binding

When N different parameters are estimated while P simulation engine processes are employed, then these P simulation engines have to communicate with N global control processes. As mentioned, AKAROA's PSM is an extension of the Remote Procedure Call (RPC) subsystem ([BIRR84], [BRIA90]) to serve AKAROA PSM's interprocess communication needs. When initiating communication between a (simulator, global control process) pair, eight pieces of PSM program are involved: simulator_stub, simulator_RPC_Runtime, Directory_Central_RPC_Runtime, Director_Launcher_stub, Directory_Central_server, plus global_control_runtime, global_control_stub, and global_control_server.

A simulation engine is an activation of a simulation program constructed with the Control, PSM (and Build) components. Each engine is executed independently on a separate processor or computer, and when execution of the program reaches the following statement added by the user for parallelizing simulation:

result = sa.processnewobs(new_observation, parameter_number) ;

a new call to the global control process responsible for gathering local estimates from the simulations engines of the corresponding parameter maybe initiated (refer to chapter 4 for a description of the SA-PTS MRIP methodology, and Chapter 5 for the user programming interface to the implementation of SA-PTS in AKAROA). This occurs whenever a local spectral analysis checkpoint for parameter identified by parameter_number has been reached.

The caller-engine, i.e. the simulator-stub of the simulation engine making the call, generates a locate-request-datagram, then tells its RPC-runtime to transmit it to the well known Directory_Central process, specifying its (the caller-engine's) machine address and port number, the type of call, and the type and the instance of global-estimation control process it wishes to be connected with. With SA-PTS the type would be a global spectral analysis control process. The instance identifies which of the various global spectral analysis processes it needs, i.e. the process corresponding to the parameter being estimated.

The Directory_Central, once receiving the caller-engine's datagram, searches its Active-Control-Process (ACP) Database for a process of the requested type and instance. If found it transmits a datagram to the caller-engine with the location of the desired control process. Otherwise, the Directory_Central launches a global control process (either on its host machine, or on a remote machine through a Local_manager process), updates its ACP Database, and transmits a datagram to the caller-engine with the newly created location of control process. The caller-engine next transmits a datagram with the appropriate data (parameter identification number, the current value of its point estimate and its precision, the length of initial transient, and the total number of observations generated so far) to the located instance of the global control process. Upon receiving the datagram from the caller-engine, the global control process updates its global_estimation (GES) and Registered_Simulation_Engines (RSE) databases and computes a combined estimate of the mean and its relative precision. If the desired level of precision is achieved, a STOP datagram is returned to the caller-engine, otherwise a CONT (continue) datagram is returned. This process is depicted in Fig. 6.4.

Subsequent global_estimation calls are made directly to the global_control process, without going through the Directory_Central.

Simulation Launch Sequence

A user runs a simulation using the "akkey" command on the user's workstation (see section **??**n page 100). The parameters to the akkey command include the name of the simulation and any parameters it might require, and possibly the value of P, i.e. number of simulation engines that should be employed. The initiation of a simulation involves 3 main tasks.

1. The akkey process (running on the user's machine) makes a request for running a new simulation (LREQ) to the Directory_Central process (possibly running on another machine).
2. Upon receiving the request, Directory_Central process searches its Registered-AKAROA-Machine (RAM) to find P machines in the AKAROA domain that are most suitable for hosting simulation engines. Next, Directory_Central sends a simulation engine launch request to Local_Manager process at each of the chosen machines.

Figure 6.4: Simulation Engines — Control Processes Binding

3. Upon receiving a simulation engine launch request, each Local_Manager would create a new simulation engine process, and report back to the Directory_Central.

These tasks are performed in the following way.

The akkey-main, akkey-stub, and akkey-RPC-runtime execute on the user's machine. When the user runs a simulation with akkey, akkey-main invokes a akkey-stub function for building a Launch Request (LREQ) datagram. This datagram would hold a specification of the type of service required, and a variable number of arguments. For akkey, the service type would be Launch Request (LREQ). The arguments would include the name of the SE executable and parameters, and the priority and hostnames of machines requested if specified by the user using the akkey option. When the LREQ datagram has been assembled, akkey-stub asks akkey-RPC-runtime to transmit the datagram to the machine that hosts the Directory_central process. akkey-RPC-runtime would then add a call identifier to the datagram, transmit it to the directory_central process at its well known address (machine address and port number), sets an alarm. and blocks, waiting for a proper response. If a response corresponding to this call has not been received when the alarm expires, akkey-RPC-runtime timesout and retransmits the LREQ datagram [1].

On receipt of the LREQ datagram, the RPC-runtime of Directory_central's host machine notes the call identifier in the datagram, then passes the datagram to the Directory_Central-stub. The Directory_Central-stub unpacks it and makes a function call invoking the Directory_Central-server procedure for simulation launch. Once invoked, the Directory_Central-server's simulation launch function searches its Registered-AKAROA-Machine (RAM) Database for the most appropriate machines for hosting the simulation processes. Having selected P suitable machines, it makes one Simulation Engine Launch Request (SELR) call to each Local-manager-server of the selected machines. Each call by the Directory_Central-server is handled through the Directory_Central-stub and Directory_Central's RPC-Runtime (on Directory_Central's machine), and the Local_Manager's RPC-Runtime and the Local-Manager-stub (on the Local_Manager's machine).

Following the receipt of a SELR datagram, the Local-manager-stub unpacks it and makes a Simulation Engine Launch call to the Local-manager-server. Local-manager-server folks, with its child executing the requested Simulation Engine through an exec local kernel call. The parent then returns a SELA datagram to the directory_central process (through its Local-manager-stub and RPC-runtime), to inform Directory_central that the simulation engine was successfully launched on its machine.

After making each SELR call, the Directory_Central-server updates the entry of the called machine in its RAM database, to note that machine is executing a simulation engine. Having made P SELR calls the Directory_Central-server re-

[1] in the current implementation, up to five retries are allowed before akkey-RPC-runtime gives up and returns a failure indicator

turns a success indicator together with the number of machines running simulation engine processes, to its Directory_Central-stub, and the results are packed into a datagram together with the call identifier that accompanied the LREQ, and passed back to the blocked akkey process in the user's machine. There they are unpacked and the akkey-stub returns them to akkey-server. Next, akkey-server informs the user of a successful launch and the number of machines engaged (for SE execution), and then blocks, waiting for simulation completion.

Meanwhile, each of the simulation engines (typically on separate machines in the network) created by the SELR calls will make a Generator Request (GREQ) remote function call when their execution encounters the declaration of the SPECTRALANALYSIS object. The purpose of GREQ is twofold. This GREQ call is made by the constructor of SPECTRALANALYSIS, which in turn invokes the appropriate function in the simulation-engine-stub for assembling a GREQ datagram, then tells simulation-engine-RPCruntime to transmit the datagram to the Directory_Central process's machine and port. On receipt of a GREQ packet, the Directory_Central process obtains the first unassigned pseudo random number generator searches from its generator database (each simulation engine must use a different random number generator, and such that the sequences of numbers created by each generator are not correlated). Directory_central would then assign it to the calling simulation engine by sending the seed defining the generator as the return value of the GREQ call. This return value is packed into a Generator Return (GRET) datagram by Directory_Central-stub, and then transmitted to the calling simulation-engine's machine by Directory_Central-RPCruntime. Secondly, Directory_Central would update its Active Simulation Engine (ASE) database with an entry for the calling simulatin engine process. Each entry include information on the simulation's location and status.

Thirdly, if this is the first GREQ packet received during the current simulation run, then a Directory_Central sets a timer for measuring (real) simulation run time. The time taken for the parallel simulation run is recorded in default reports produced by AKAROA.

The transactions between the akkey, directory_central and local_manager processes in this launch phase are summarised in Fig. 6.5. The databases maintained by the Directory_Central, their conceptual scheme, and their physical implementation are depicted in Fig. 6.6.

Figure 6.5: AKAROA process interactions during a simulation launch.

Method for Terminating a Parallel Simulation

The problem of deciding when estimates of all parameters have reached a desired level of precision becomes interesting when more than one parameter is being estimated during one simulation run. Each global control process maintains only information pertaining to the parameter it is responsible for estimating. Therefore, no global_control process can decide when all estimates are at required pre-

Figure 6.6: a) Conceptual Scheme, and b) physical data structures of Directory_central's databases

cision, based on their local knowledge alone. Even the ASE database (maintained by Directory_Central) and the RSE databases (one version maintained by each global_control process), cannot be assumed to agree. For instance, M simulations may have registered with Directory_Central's ASE (Active Simulation Engine) database by making a GREQ remote call, see above. If data generated by N of these engines ($N < M$) is sufficient to satisfy precision requirements of estimates whilst the remaining M-N engines are still executing the simulated system's transient stage then the stopping criterion has been met even though at least M-N engines have not binded with the global_control processes, and the ASE database would differ from the RES ones.

The solution implemented in AKAROA, bearing in mind the need for efficiency, transparency and fault tolerance, involves joint efforts by the global_control processes, simulation_engines, the directory_central, and the local_managers. Each simulation is responsible for maintaining its copy of a progress status board. This status board has P squares, one per parameter being estimated. When an engine reaches a checkpoint for parameter p_i, and makes a global_estimation (GES) call, the value returned from the (remote) global_control process indicates whether the global estimate of p_i has attained required precision. If, after examining the return value, the engine finds that p_i has being estimated to required precision, then the engine puts a check in the ith box of its status board. By examining its status board each time after updating it, an engine can decide whether all parameters are at required level of precision, and hence whether the stopping criterion has been met.

If an engine, upon inspection of its statusboard, finds that all boxes are checked, it will initiate the termination sequence by making a Stop REQuest (SREQ) call to directory_central. When directory-central receives the SREQ datagram, the stop_request procedure of directory-central-server will be invoked. stop_request firstly steps through its ASE database. If the entry does not correspond to the simulation engine that made the SREQ call , then it makes a Kill REQuests (KREQ) call to the local_mamager on the machine that hosts the simulation engine process, passing the engine's pid as an argument. Upon reception of a KREQ datagram, the local_mamager will send a STOP signal to the (simulation engine) process with the pid, terminating the process. Hence, with M engines in ASE database, M-1 KREQ calls will be made.

After stopping all but one simulation engines, the directory_central terminates all control processes using the remote local_managers in a similar manner. Finally, directory completes the simulation by terminating the (blocked and remote) akkey process by means of a KREQ call. This termination procedure is summarised in Fig. **??**.

This procedure is correct in the sense that if termination conditions are met, then this protocol guarantees that the termination condition would eventually be detected. Refer to Fig. 6.8 for an example of such a scenario. Notwithstanding, this solution is sub-optimal: achievement of termination conditions may not be immediately recognised ! Alternative distributed termination protocols considered are presented in the next section.

Figure 6.7: Simulation termination sequence, when estimating four parameters during the run, using three machines to execute simulation engines.

Figure 6.8: Delayed termination: conditions for termination has been met, but has not been detected yet by any SE

6.2.4 Implementation Decisions

Dynamic Binding. The tasks of locating and connecting a simulation-engine to a given global-control-process is done at run-time and is handled automatically by AKAROA's PSM. Thus, simplicity of use, one of our major aims, is achieved. All details of parallel simulation such as machine addresses on which simulation processes exist, locations of the desired control processes for communication, and ascertaining their corresponding port numbers are hidden from the user. Another consequence is that the processes (simulation engines and global-control processes), their number, and the machines for their execution may be decided after program compilation and linking. These decisions can therefore be made to suit the current availability and load of the computers on a network. In fact simulation processes can be created during parallel simulation runs, and they will automatically be accommodated by AKAROA's interprocess communication sub-system of PSM.

Internet Domain Datagram Type Sockets. We implemented intermachine interprocess communication using UNIX sockets over the Internet domain and of type datagram. The datagram type was chosen for its efficiency, and because it would not limit the number of (parallel) simulation processes. Each datagram message is addressed individually. Operations with Internet datagrams are fully file-less, and even the initial rendezvous is done without creation of an i-node (unlike X-Windows). The price of the performance gain from using the datagram style of communication is that datagrams may be lost, or delivered out of order.

Fault Detection. RPC-Runtime of the simulation-engines, Directory_Central, and global-control-processes employ a time-out with a scheme of retransmissions for detecting problems with lost datagrams. Currently five retries are allowed, after which it returns ERRS to the associated stub, which returns that as the result of the function call, indicating a fault.

Non-Idempotent Operations. Idempotent operations are those which if repeated have the same effect as if they were performed once. With an automatic retry scheme there is a possibly that a sender of a datagram erroneously sends a datagram more than once. For instance, if response datagrams were lost, but the sending process's datagrams were delivered correctly to the receiver, then the sender would retry sending the same datagram several times up to the maximum number of retransmissions allowed.

To guard against erroneous actions from processing duplicate datagrams, we let RPC-Runtime add time-stamps to the control field of each datagram. For example, a global_control process can compare the timestamp carried by the current GES request datagram with the previous timestamp from the same engine. If the current timestamp is equal to or less than the previous one, the GES call can be discarded as a duplicate (after returning the same result). The other purpose of the timestamp is to allow the caller (e.g. the simulation engine) to determine that the received datagram is truly the reply to its current call.

We also considered the possibility of using sequence numbers for the same role, instead of time stamps. To detect duplicate datagrams, the callee keeps a record of the previous sequence number sent by a each calling process. When the next request datagram is received, the callee compares its sequence number with the previous one from that caller, and concludes that it should be discarded as duplicated if the number is smaller than or equal to the previous one. However a problem arises when the caller crashes and then restarts. The callee process (typically on another machine) would not know about the crash. When the next datagram is sent to the callee from the restarted caller, it would (erroronously) be dismissed as a duplicate or 'out of date'. As an illustration if the machine executing akkey crashed and restarted, subsequent calls from that machine to directory_central may be mistreated as duplicate, if the port number assigned to the restarted process was the same as the pre-crash one.

Fault Tolerance. Crashes of computers running simulation engines are tolerable. They only degrade the speed at which the required precision of estimates is obtained. Recovery to 'full speed' by launching additional simulation-engines on other machines can be easily implemented, and is under consideration. Also, once binding between simulation-engines and global-control-processes have been completed, the Directory_Central process could be killed without disrupting the progress of the simulation.

Termination Options. It is not an immediate consequence of our goals that we should place the responsibility for termination condition detection on simulation engines. Intuitively the global_control processes would be the obvious first place, since they would be the first to know when a parameter has been estimated to a desired level of precision. Nevertheless, as discussed, each global_control process only knows the progress of estimating one (its own) parameter. Indeed, for simulations where only one parameter is being estimated, the (single) global_control process is the ideal agent for detecting conditions for termination. When estimating P parameters, however, we must have some way of allowing the P control processes to reach a common agreement. The following candidate solutions were explored:

1. Provide a common status-board for control process use. The status board would have P squares when P parameters are being estimated. The ith control process puts a check on the ith square in the status board when the estimate of the ith parameter has achieved required precision. Furthermore, after entering the check mark on the status board, the ith process also see whether all P boxes of the statusboard has been checked. If all P boxed were checked, then the ith control process can assume that the stopping criterion has been met, and 'give the alarm' to directory_central for initiating appropriate process termination procedures. This protocol is illustrated in Fig. 6.9.

Figure 6.9: Common status-board based distributed termination protocol. Notice that writes and reads on the shared status-board by different GC processes may be interleaved in any order.

If the AKAROA processes all executed on one shared-memory multiprocessor, then this 'status board' solution may be ideal. Nevertheless, when using a network of workstations, implementation of a common 'status board' may be costly, due to the absence of a shared address space. For this reason we dismissed using any solutions based on the use of some form of common 'status board' early on.

2. Place the responsibilities of global_estimate computations for all P parameters on one global_control process. This control process can easily test whether all (P) estimates are at required precision, and the process can therefore detect whether the condition for termination has been satisfied, see Fig. 6.10. A variation of this solution is the use of P lightweight processes for global_estimation tasks, one per parameter being estimated. Although simple, both solutions are unsatisfactory because they would not provide the concurrency that is possible and desirable with a multi-machine network. Furthermore, the processing load placed on the common global control process would increase with P. Slower response time by the global corneal process means longer waiting (idle) times are experienced by simulation engines during each global_estimation (remote) call. Let us note that even if it was (somehow) possible to distribute execution of P light weight control processes on P machines, we would be faced with the same distributed agreement problem, due to the expense of simulating a shared address space. For these reasons, we will focus on solutions based on the use of P concurrent control processes.

Figure 6.10: Monolithic global control process based termination protocol. It is correct but cannot take advantage of the opportunity to compute global estimates of different parameters in parallel..

3. Global_control_process group broadcast. An alternative solution to the distributed agreement problem is for each global_control process to ask all other control processes if their estimate has achieved required precision, when its estimate first achieves the desired level of precision. Hence when global_control process G_i for parameter p_i receives a local estimate from a simulation, and then finds that the revised global estimate of p_i is at the required level of accuracy, it will ask all G_j (j=1, 2, ..., P; $j \neq i$) other control processes if their estimate have achieved the necessary level of precision too. If so then the termination condition has been met, and the simulation could be stopped, see Fig. 6.11. This solution is correct. Also it is efficient in the sense that the termination condition would be detected at the earliest moment. Notwithstanding, it is too costly to implement because firstly, each control process does not know the location and identity of the other (P-1) control processes, and secondly a large number of questions have to be exchanged between control processes before an agreement could be reached. The first problem can be resolved by directing (P-1) questions by a control process to other control process through directory_central. After receiving such a 'Reached_precision ?' request from a control process, the directory central would then query all other control processes in its GES database, then return their answers to the caller control process. Unfortunately, the overhead of this approach includes the transmission of $2P(P$-$1)+2$ datagrams. Also, every control process would be inconvenienced by having to responding to up to (P-1) 'Reached_precision ?' questions. A further drawback is that control processes which have already completed their global_estimation task must wait around to answer such questions, instead of terminating immediately.

Figure 6.11: Global Control Process group broadcast based termination protocol (Only messages associated with a broadcast by global precision control process GC1 are shown in this diagram).

4. Employ a referee. Appoint a PSM process as referee who is responsible for determining termination. Let each control process report to the referee when its estimate first achieves the required precision. When this referee receives the Pth 'reached_precision' report, it knows that all P parameters have been estimated to required level of accuracy. It could then initiate the termination sequence. This solution is both efficient and correct, and an attractive alternative to the one implemented in AKAROA. Overhead is quite low if we appoint Directory_central as the referee, as shown in Fig. 6.12. The cost would be $2P$ datagram transmissions (for the 'reached_precision' calls), plus one status board maintained by Directory_Central. In contrast, the solution implemented in AKAROA requires no datagram transmissions, but needs M status boards, one per simulation engine. However, as discussed it is sub-optimal, in that recognition of the termination condition may be delayed. In the end, the primary reason for our choice is that delegating the task of termination detection to simulation engines more naturally matches ordinary (single machine) sequential simulation semantics. In an ordinary simulation run, the termination condition would be detected by the Spectralanalysis object (during a processnewobs call). With our chosen implementation, the termination condition would be detected by the SPECTRALANALYSIS object, and be seen by the programmer (through the return value of the call) in the same way. Hence the semantics of Spectralanalysis (for single machine) and SPECTRALANALYSIS are identical. Full parallelism transparency is therefore neatly maintained.

Figure 6.12: Common referee based termination protocol (Here the status board can be implemented using a countdown counter)

6.3 Performance Evaluation

Dynamic properties of AKAROA, and the quality of SA-PTS estimators, were tested in a series of 1600 bench-mark simulation experiments using P=1, 2, 4, and 6 processors. We considered both the geometric checkpoint distribution (default) and the constant checkpoint distribution versions of AKAROA. Our benchmarking environment, the performance measures we used, and the results of our performance studies are reported in this section.

6.3.1 Benchmarking Environment

Initial studies of AKAROA's performance were done on a local computer network (a multiprocessor SUN Server with two SPARC CPUs, various SUN 4 and SUN SPARC workstations) connected by 10Mbps Ethernet. Apart from the obvious differences in processing power between the workstations available for our investigation, none of the machines were dedicated to AKAROA's use. Specifically, they were used concurrently by our stage 3 students (numbering approx. 55), honours class (14), research students, and lecturers of the Department of Computer Science. Furthermore, the SUN server was used by several other departments in addition to ours. Our approach to evaluating AKAROA in this environment, is to conduct all single processor (P=1) simulation experiments using our fastest machine during low load periods (at night), whilst executing multi-machine experiments with AKAROA using a mix of the fast and lower rated workstations during periods when higher competition for their use is expected. In this manner, we expect our results for speedup by AKAROA to be on the pessimistic (and hence safe) side. For added safety, all processes of AKAROA were executed at lower priorities for $P>1$ cases than for single processor runs.

Main performance measures considered were: real time speedup (the ratio of time needed to achieve an estimate at a given precision level on a single processor to the time required for achieving it on P processors), CPU-time speedup (the ratio of CPU-time needed to produce an estimate at a given precision level on a single processor to the CPU-time required for achieving it on P processors), and coverage (the frequency with which the confidence intervals produced by the SA-PTS method contain the true parameter at a given confidence level). Analytically, we can write

$$speedup(P) = \frac{simulation\,time\;on\;one\;processor}{simulation\,time\;on\;P\;processors} \tag{6.1}$$

$$CPU - time - speedup(P) = \frac{CPU\ time\ on\ one\ processor}{average\ CPU\ time\ on\ P\ processors} \tag{6.2}$$

$$coverage(P) = \frac{no.\ of\ P\ processor\ experiments\ giving\ a\ CI\ that\ enclosed\ the\ true\ parameter}{total\ number\ of\ experiments\ using\ P\ processors} \tag{6.3}$$

The speedup measures gauge AKAROA's potential for delivering results faster over simulations executed on a single uniprocessor machine. speedup(P) represents the reduction in real time (as observed by the user) achievable through the use of P processors to execute the simulation. Thus speedup(P) accounts for the overhead incurred in parallel processing, including time required for process creation, management, and inter-machine interprocess communication, as well as the delays caused by non-AKAROA processes of other users which shares the machines we used. In comparison, CPU-time-speedup measures improvement in terms of reduction in computation time per machine engaged in simulation. Thus this measures speedup attainable if IPC, and the effects of non-AKAROA processes were negligible.

Whereas the speedup indexes report AKAROA's speed in producing estimates to a level of precision desired by the user, coverage(P) measures the quality of the resulting estimators when obtained using P workstations. All presented results were obtained during steady state simulations of $M/M/1/\infty$ queuing systems with traffic load ρ=90%; each experiment was repeated 200 times. The parameter estimated by simulation is the mean delay experienced by a customer in the system. In all experiments, a level of precision of $\pm 5\%$ or better was required of the final estimates. The level of confidence required was 95%. This means that the half width of the confidence intervals for the mean delay produced by each experiment should be no greater than 5% of the estimated value, and that the confidence intervals produced should be correct (i.e. contain the true value of the expected delay) in 95% of experiments. Much is know analytically about the $M/M/1/\infty$ system, including the true value of the expected delay, hence the correctness of the confidence intervals for the delay obtained by our simulation experiments, and hence an estimate of the coverage could be obtained.

Geometric checkpoint distribution (default) and constant checkpoint distribution options are supported by AKAROA. Intuitively, geometrically distributed checkpoints would bias speedup results in favour of sequential experiments, since less checkpoints are expected to be reached than parallel simulations, so less computation. Also in parallel simulations, communication between processes on different machines would be performed at each checkpoint, unlike the sequential simulations where interprocess communication is never needed. On the other hand, since parallel simulations result in more checkpoints being reached, and that the precision of estimators is computed at each checkpoint, we would expect that parallel simulations would stop at a level of precision closer to the minimum requirement than a corresponding sequential simulation which we expect to be more likely to 'overshoot', stopping when the precision of estimates has exceeded the target level. These competing effects cloud our ability to apply intuition in deducing whether the geometric checkpoint option favours simulations executed sequentially or in parallel, and consequently it is hard to see in advance its effects as a function of the degree of parallelism employed. We considered both the geometric checkpoint distribution (default) and the constant checkpoint distribution versions of AKAROA.

6.3.2 Results of Performance Studies

Real time speedup of simulations achieved with AKAROA as a function of the number of workstations used, are reported in Fig. 6.13 and 6.14 for geometrically and uniformly distributed checkpoints respectively. For these experiments, we used the fastest workstation for running the non-parallel simulation experiments (P=1 cases), and a mix of slower workstations for simulations in MRIP scenario ($P > 1$). Further, we lowered the priority of simulation processes engaged in parallel simulations (to accommodate other users of the network), while conducting the non-parallel simulations at the highest priority level. Thus, our results for real time speedup reported here are very conservative. Nevertheless, Figs. 6.13 and 6.14 clearly show that substantial, near linear, speedups were achieved in the case of geometrically distributed checkpoints, and practically linear speedup in the case of uniformly distributed checkpoints, suggesting the potential of AKAROA as a package for parallel steady-state simulations. One can also see the advantage of using uniformly distributed checkpoints with SA-PTS, at least in studying relatively simple simulation models.

The CPU-time-speedup (normalized to the average time to generate an observation) are presented in Fig. 6.15 and 6.16, for geometrically and uniformly distributed checkpoints respectively. This measure differs from real-time speedup in that interprocess communication time is not included, and the effects of using slower workstations for the parallel simulations are reduced. Fig. 6.15 shows

Figure 6.13: Real Time Speedup versus the number of processors employed for running simulation replications.

Figure 6.14: Real Time Speedup versus the number of processors employed for running simulation replications (Constant Checkpoint Distribution).

both the overall mean number of observations (per replication) before stopping an experiment as well as the minimum and maximum numbers of observations per replication (averaged over 200 experiments conducted). Comparing the maximum replication lengths as a function of P, we observe that the reduction in CPU time with P workstations is greater than $1/P$, suggesting super-linear speedup ! This may be due to the fact that AKAROA uses CPU time more efficiently, and n observations generated by P workstations in parallel ($P > 1$) have higher entropy than if they were collected from a single replication.

On the other hand, let us note that the speedup of MRIP simulations is probably not subjected to Amdahl's Law or its consecutive versions [GUST88]. Namely there exists a minimum number of observations to be collected by any processor to reach its first checkpoint, say n_0. Hence if the mean run length of a corresponding simulation executed on a uniprocessor equals N (N being the number of observations needed before the estimator reached the required level of precision), the mean maximum number of processors giving a speedup is $P_{max} = N/n_0$, plus the processors used for executing the global (estimation) control processes. Using more than P_{max} processors would result only in narrower final confidence intervals, but would not increase the speedup. In our typical simulations (simulations of data communication networks and their protocols, and of processor memory interconnection networks), one may assess P_{max} as greater than 100. These prognostications are yet to be confirmed by additional experimental studies. We are looking for the possibility of conducting experiments with AKAROA on large numbers of processors in the near future.

Comparing the results presented in Fig. 6.15 with those in Fig. 6.16 one can again see the advantage of using uniformly distributed checkpoints in SA-PTS, since this strategy substantially shorten simulation runs (notice the different vertical scales used in these figures).

The average number of messages (datagrams) exchanged during a parallel simulation run as a function of P is shown in Fig. 6.17 and 6.18, for geometrically and uniformly distributed checkpoints respectively. As we see, the communication overhead grows slower than linearly with the number of communicating processors, in the case of geometrically distributed checkpoints. In contrast, the IPC overhead, as measured by the average number of messages (datagrams) exchanged during a parallel simulation run, is essentially constant for uniformly distributed checkpoints. It should be noted however, that the IPC over head when uniformly distributed checkpoints were used is significantly greater than when geometrically checkpoints were used (e.g. approximately an average of 1050 datagrams versus 135 datagrams per simulation run). This is due to the fact that when the

Figure 6.15: Speedup measured by reduction in CPU time versus the number of processors employed for running simulation replications.

checkpoints were uniformly distributed, more checkpoints are reached pre run, resulting in greater computational requirement for precision control (a local and global point and interval estimate is calculated at each checkpoint), and higher interprocess communications overheads. Another apparent observation is that the number of datagram transmissions per run seem to be relatively independent of the number of processors we used, when checkpoints were uniformly distributed. This appeals intuitively since speedup is linear, when checkpoints are uniformly distributed the IPC overhead of a simulation run would be more or less fixed. The very high IPC overhead of the uniform checkpoint option, combined with its high speedup with respect to the geometrically distributed checkpoint case, suggests the good efficiency of the IPC subsystem implemented in the PSM of AKAROA.

The results of the quality analysis of SA-PTS estimators as shown in Fig. 6.19 and 6.20, where the average relative precision of the estimate at stopping point is depicted, and in Table 6.1 and 6.2, which contain the results of our coverage analysis. When the checkpoints were uniformly distributed, the final precision of the estimates were closer to the required threshold value of the precision than when geometrically distributed checkpoints were used. This could be intuitively explained by the fact that the distances between consecutive checkpoints are smaller in the

Figure 6.16: Speedup measured by reduction in CPU time versus the number of processors employed for running simulation replications (Constant Checkpoint Distribution).

parallel simulations where the checkpoints were uniformly distributed, so more checkpoints are encountered during a simulation run. More checkpoints reached means greater more frequent calculations of estimates and their precision, so simulations are stopped sooner after enough observations have been generated to satisfy the stopping criterion.

Figure 6.17: Average number of datagrams exchanged versus the number of processors employed for running simulation replications (geometric checkpoint distribution case).

The results of the coverage analysis show that using more processors in MRIP simulations improve the coverage of results. This can be due to the fact that the mean entropy of an observation grows with P. Note that SA-PTS for $P > 1$ always appeared to be a valid method of estimation, since the confidence intervals of coverage always contained the theoretical confidence level, c.f. Table 6.1 and 6.2. On the other hand these results confirm our analysis contained in Chapter 3, that SA/HW—inspite of being the best of the methodologies we benchmarked—may be invalid for high loaded queuing systems, although the level of error is not too significant.

Figure 6.18: Average number of datagrams exchanged versus the number of processors employed for running simulation replications (Constant Checkpoint Distribution).

P	Mean	95% Confidence Interval
1	0.87000	(0.82338, 0.91662)
2	0.94500	(0.91332, 0.97668)
4	0.96465	(0.93886, 0.99044)
6	0.94000	(0.90700, 0.97300)

Table 6.1: Coverage and its confidence intervals for SA-PTS (assumed confidence level =0.95); Geometric Checkpoint Distribution

Figure 6.19: Relative precision of estimator achieved at stopping point (geometric checkpoint distribution).

Figure 6.20: Relative precision of estimator achieved at stopping point (constant checkpoint distribution)

P	Mean	95% Confidence Interval
1	0.87000	(0.82338, 0.91662)
2	0.94000	(0.907079, 0.972921)
4	0.93000	(0.894631, 0.965369)
6	0.92000	(0.882393, 0.957607)

Table 6.2: Coverage and its confidence intervals for SA-PTS (assumed confidence level =0.95); Uniform Checkpoint Distribution

Chapter 7

Conclusions

Since any stochastic simulation is a kind of a statistical experiment, estimates obtained from simulation output data (observations collected during simulation runs) have to be properly, statistically analysed. The simplest way of succeeding in this is to apply sequential precision control of estimates during a simulation run, where the precision of estimates is inferred during simulation at consecutive checkpoints, until the required level of precision is obtained. Such an approach is specially important in steady-state simulations, conducted for studying systems' behaviour after a long period of time. Unfortunately, inferring the precision of steady state estimators is difficult because the output data is usually correlated, and the simulated system typically traverses a transient phase before reaching its steady state. This problem of obtaining statistically reliable results is compounded by the fact that simulations of even moderately complex systems, especially in studies of their steady-state behaviour, often becomes computationally intensive and can require very long runs in order to obtain the required precision of final results. Excessive run-times hinder the development and validation of simulation models, and may even totally inhibit some performance evaluation studies.

In this report we have

1. Surveyed the basic problems and solutions of automatic sequential precision control of steady-state estimates (in Chapter 2).

2. Conducted comparative studies (reported in Chapter 3) of the more promising methods for automatic precision control of estimates of steady-state mean values selected by us in Chapter 2 .

3. Surveyed previous proposed methods for applying parallel and/or distributed processing for speeding-up quantitative stochastic simulations, (Chapter 4).

4. Proposed (in Chapter 4) the SA-PTS method, a novel approach to parallel simulation that employs a generalisation of the best method of sequential estimation suggested in Chapter 3.

5. Implemented and analysed the performance of AKAROA, a parallel simulation package for the estimation of mean values in steady-state simulation experiments following the SA-PTS methodology proposed in Chapter 4. The design goals and user interface of AKAROA were reported in Chapter 5, and the architecture, implementation and benchmarking of AKAROA were reported in Chapter 6.

Thus the main findings of this report have been used to synthesise AKAROA, a parallel simulation package built primarily for conducting quantitative steady state simulations in Engineering and Computer Science. One of AKAROA's primary aims is to obtain statistically reliable estimates of steady state parameters by simulation (e.g. the mean packet delay, or throughput), and to do this with minimum user effort, and within a practical timeframe. Obtaining statistically reliable estimates is specially difficult because classical statistical techniques cannot be adapted for controlling the precision of estimates obtained during a simulation run. As mentioned, this is because observations collected from the simulated processes are usually correlated, and the processes of interests usually pass through a transient phase. A secondary aim is that AKAROA should be sufficiently flexible to be applicable to the performance analysis of a variety of telecommunication systems. Some interesting aspects of AKAROA are

- Precision analysis and control functions are performed automatically through an AKAROA object that can be easily attached to a simulator once it has been written. Thus, to use AKAROA's sequential precision control services a user simply declares an object for output data analysis. Its member function responsible for precision control is later called whenever a new observation is generated. The function accepts the value of a new observation as parameter and returns one of two values that either orders the simulation to be continued (desired precision of estimates has not been achieved) or to be terminated (all estimates reached the required level of precision). All complexities associated with the statistical analysis of output from autocorrelated processes, each typically passing through their own non-stationary phases, are invisible to the user.

AKAROA's user programming interface for parallel simulator development is semantically identical to that described above for normal non-parallel simulation; only the type of object that needs to be declared is different. Thus, when programming a simulator, users do not even need to be aware of the existence of multiple (parallel) simulation and control processes during simulation run-time, since their creation, location (machine and port addresses), co-operation, and inter-machine interprocess communication, are hidden from users. The syntax for object declaration and for calling the object's member functions, as well as the meaning of the values that may be returned by them are also identical to that described above for the non-parallel case, yielding a programming interface that features transparent parallelization from users' point of view, both in the semantic and syntactic sense.

- A novel methodology for parallel stochastic simulation, called Spectral Analysis in Parallel Time Streams (SA-PTS), is used by AKAROA for obtaining the required precision of final results.

  SA-PTS was proposed by us, following our survey of options for applying parallel and/or distributed processing in quantitative stochastic simulation in Chapter 4, and basing on our previous experience in non-parallel stochastic simulation and exhaustive comparative studies of various sequential estimators (reported in Chapters 2 and 3). SA-PTS is a generalisation of sequential Spectral Analysis (SA/HW) in its HW version, i.e. the method for precision inference of estimates for simulations executed sequentially (on uniprocessors) evaluated in Chapter 3 to be the best of the candidate methods.

- AKAROA automatically transforms a simulator that was written for a uniprocessor into one suited for parallel execution on a multiprocessor, or a heterogeneous network of workstations.

- At run-time, AKAROA presents a network of workstations as a single logical (virtual) machine to the user at runtime. With AKAROA, a user can start or stop a parallel simulation as if it was a single process running on one workstation. Like a single machine, a user can also schedule the running of different simulations one after the other using a shell script. All aspects of distributed simulation, including distributed process creation, the binding of simulation engine processes with global (precision) control processes, inter-machine interprocess communication, fault detection, and distributed agreement and termination are performed invisibly by AKAROA.

- During the installation of AKAROA (normally done just once), its parallel simulation manager (PSM) allows the user to specify the machines in the network that AKAROA is allowed to use for simulation execution (the collection of registered machines are called the AKAROA domain). The AKAROA domain may be changed after installation. The user can also divide the workstations in the AKAROA domain into multiple classes to reflect their political/ownership status with respect to the user. When AKAROA selects machines for hosting simulation processes, it takes into account their membership class.

- Dynamic binding between distributed processes co-operating in execution a simulation is employed by AKAROA. This permits machine selection, and the allocation of machines (processors) to simulation processes to be done at runtime instead of compile time. Machine failures, or the addition or removal of machines from the AKAROA domain can therefore be exploited without recompilation.

- Load Balancing can be realised to a great extent through 1) the ability of AKAROA's PSM to assign processes to machines during runtime, and to set their 'nice' priorities based on their membership class, and 2) many AKAROA processes can be suspended without hindering (block) the progress of AKAROA processes on other machines.

- Crashes of most machines which hosts AKAROA processes can be tolerated.

AKAROA has the following limitations :

- All AKAROA simulation engines must traverse their own warm-up period. If the steady state runlength is shorter than the transient length, then the speed up from the use of multiple processors could be degraded. The steady state runlength of each simulation engine is inversely proportional to the number of processors (workstations) employed. Thus for processes which initially goes through a long transient period, there may be a limit on the number of processors that can be used, beyond which a diminishing return effect with respect to speedup may be encountered.

- On a heterogeneous network, if the simulated processes traverses a long transient period before reaching steady state, and if one workstation is significantly faster than the others then it may reach steady state much earlier than others, and produce sufficient observations before most others achieved steady state. In such a case, the use of the other workstations may not contribute to speeding up the simulation task.

- The speedup of simulation execution under SA-PTS is probably not subjected to Amdahl's Law or its consecutive versions [GUST88], since there exists a minimum number of observations to be generated by each simulation engine to reach its first checkpoint, say n_0. Thus if the mean runlength of a corresponding uniprocessor simulation equals N (i.e. N observations are needed for reaching the required level of precision), the (mean) maximum number of simulation engines (each running on its own processor/workstation) giving speedup if $P_{max} = N/n_0$.

- The relative rate of network use increases with the number of machines employed to run AKAROA processes. Recall that according to our benchmarks the number of datagram transmissions between processes on different machines is approximately constant, independent of the number of machines employed. Hence as we increase the degree of parallelism by using more machines, the simulation execution time would be reduced (speed up), and accordingly the rate of datagram transmissions also increases. By this reason we may encounter a network bottleneck, as the number of machines used is increased.

However, basing on our benchmark experiments, we did not appear to have reached any of these limits (most machines used were interconnected by 10Mbps Ethernet).

All modules of AKAROA have been completed and, in addition to executing our benchmark simulations, AKAROA has also been successfully used to construct parallel simulators and conduct simulations of WDM local networks, DQDB networks, ATM buffer overload control mechanisms, CSMA/CD local area networks, as well as standard queuing systems.

At present, AKAROA can only be use for precise estimation of (multiple) mean values, and higher moments about the origin[1]. Its performance evaluation is continued. In the nearest future its statistical abilities will be extended by including steady-state estimation of quantiles, and functional relationships (applying response surface methodology) and their precision. Whereas the package is primary intended for use in steady-state stochastic simulation, it can be easy

[1]Central moments can therefore be estimated, as a function of estimates about the origin

extended to non-steady-state simulations. Tests of speedup, although so far involving less than ten processors, have given very encouraging results. In the nearest future experiments in a LAN with a larger number of computers are planned. Our intention also is to augment AKAROA with a graphical interface, for visualisation of various stages of simulation, sequential data analysis and for presenting the final results. Another open option is to explore the suitability of additional pesudo-random number generators for intergration into AKAROA, on the basis of their computational complexity, individual quality, and the extent to which their sequences are mutually uncorrelated.

Appendix A

References

[PAWL88] Pawlikowski, K., and M. Asgarkhani. "Sequential Procedures in Simulation Studies of Satellite Protocols". Proc. ITC'12, Torino, Italy, 1988, vol.6, 4.3B.3.17.

[PAWL90a] Pawlikowski, K., C.Karthikeyan and H. Sirisena. 1990. "Batch Means Techniques in Steady State Simulation of DQDB Network". Proc. 5th Australian TeletrafficSeminar (Melbourne, Australia, Nov.1990), Telecom Australia Research Laboratories Press, 1990, 1-10. [PAWL90] Pawlikowski, K. "Steady-state Simulation of Queueing Processes: A Survey of Problems and Solutions". ACM Computing Surveys, no.2, June 1990, 124-170

[PAWL91] K.Pawlikowski and V.Yau. "Independent Replications versus Spectral Analysis of Output Data in Steady-State Simulation of High Speed Data Networks". Proc. 6th Australian Teletraffic Research Seminar (Wollongong, Australia, Nov. 1991), Univ. of Wollongong Press, 1991, 322-330.

[PAWL92] K.Pawlikowski and V.Yau."An Automatic Partitioning, Runtime Control and Output Analysis Methodology for Massively Parallel Simulations". Will be published in Proc. European Simulation Symp., ESS'92, , Dresden, Germany, Nov.1992.

[PAWL92] K.Pawlikowski, and V.Yau, On automatic partitioning, runtime control and output analysis methodology for massively parallel simualtions, Proc. European Simulation Symp., ESS92 (Dresden, Germany), Nov.1992, pp.135-139.

[PAWL93] K.Pawlikowski and V.Yau. "Methodologies for stochastic steady state simulations of communication networks," Telecom NZ final research report, 1993.

[PAWL94] Pawlikowski K., V.Yau and D.McNickle. "Distributed stochastic discrete-event simulation in parallel time streams". Proc. 1994 Winter Simulation Conference, IEEE Press, pp. 723-730

[YAU93] V.Yau and K.Pawlikowski, AKAROA: a package for automatic generation and process control of parallel stochastic simulation. Proc.16th Australian Computer Science Conf., ACSC93 (Brisbane Australia), Feb.1993, vol.A, 1993, pp.71-82.

[YAUP92] Our IEEE TENCON ntcd/se paper

[YAU96] V.Yau, "Polarized Independent Replications : A Method for Runlength Control of Stochastic Steady State Simulation Experiments", in preparation.

[PAWL94] Pawlikowski K., Yau V., and McNickle D., Distributed stochastic discrete-event simulation in parallel time streams, Proc. 1994 Winter Simulation Conference, IEEE Press, pp.723-730

[YAU93] Yau V. and Pawlikowski K., AKAROA: a package for automating generation and process control of parallel stochastic simulation, in proc. of the Sixteenth Australian Computer Science Conference, ed.G.Gopal et al., Australian Computer Science Communications, 1993, pp71-83.

[YAU92] Yau V. and Pawlikowski K., On automatic partitioning, runtime control and output analysis methodology for massively parallel simulations, Proc. European Simulation Symposium ESS 92, pp.135-139, Dresden, Germany, Nov 1992.

[PAWL92] Pawlikowski K., Yau V., An empirical comparison of sequential estimators for output data analysis in steady state simulation of high speed data networks, in proc. Operations Research Society Conference92, pp175-177, Christchurch, New Zealand, 1992.

[YAU92] Yau V. and Pawlikowski K., A class of protocols for heavy loaded multiple-channel local area networks, in proceedings IEEE/ACM International Conference on Communications ICC92, pp 742-748, July, 1992, in Chicago, U.S.A.

[YAU92] Yau V and Pawlikowski K., Improved Nested-Threshold-Cell-Discard buffer management mechanisms, in proc. IEEE Int. Conf. on Computers, Communications, and Automation, TENCON92, (Melbourne, Australia, Nov1992), IEEE Comm.Press, 1992, pp 820-824.

[YAU92] Yau V. and Pawlikowski K., ATM Overload Control: Nested Threshold Cell Discarding with Suspended Execution, in proc. of the Australian Broadband Switching and Services Symposium, ABSSS92, pp 689-706, Melbourne, 1992.

[PAWL91] Pawlikowski K., Yau V., Independent replications versus spectral analysis in steady-state simulation of high speed data networks, in proc. ATRS91, Nov. 1991, Wollongong, Australia, pp 182-190.

[YAU96] Yau V., and Pawlikowski K., "A Conflict-free Traffic Assignment Algorithm using Forward Planning," IEEE INFOCOM'96, vol.3.

Pawlikowski K., McNickle D., and Yau V., Object-oriented model construction, and automated distributed simulator generation and output analysis , Final R&D report, Australia Overseas Telecom Corporation (AOTC Telstra), Con. 7314, Also, technical report no. COSC 02/93, Dept. of Computer Science, University of Canterbury.

Pawlikowski K., and Yau V., Methodology for stochastic simulation for performance evaluation of data communication networks, Final Report for Telecom Corporation of New Zealand, Wellington, New Zealand. Also Technical report no. COSC 03/93, Dept. of Computer Science, University of Canterbury.

Yau V. "Polarised Independent Replications: An alternative approach to estimating steady state parameters by simulation in the presence of an initialisation bias", in preparation.